UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

Trabalho de graduação

Avaliação de idiomas baseados em AspectJ para implementar *features* com *binding times* flexíveis em Linhas de Produto de Software

Autor: Rodrigo Cardoso Amaral de Andrade
Orientador: Paulo Henrique Monteiro Borba

Recife, Novembro de 2009

Resumo

Linhas de Produto de Software (LPS) são famílias de produtos de software que compartilham um conjunto de artefatos em comum. Elas são desenvolvidas a partir de uma arquitetura em comum, e compartilham alguns artefatos reusáveis.

No contexto de LPS é importante definir *feature*. *Features* são incrementos na funcionalidade de um programa, são as unidades semânticas pelas quais programas dentro de uma família ou LPS podem ser diferenciados e definidos.

Outra definição importante é o conceito de *binding time*. Nesse contexto, o *binding time* de uma *feature* é o momento em que é decidido a inclusão ou exclusão da mesma em um produto. Definir uma LPS em que as *features* possuem *binding times* flexíveis é importante, pois, por exemplo, para ambientes que possuem limitações de recursos, não é aconselhável ter *binding time* dinâmico. Para implementar *features* em uma LPS, vários mecanismos podem ser utilizados, tais como Programação Orientada a Aspectos, Padrões de Projeto, Arquivos de Configuração, etc.

Para permitir essa flexibilidade, alguns idiomas baseados em AspectJ foram criados. No entanto, algumas limitações nestes idiomas foram identificadas neste trabalho, tais como duplicação de código.

Assim sendo, este trabalho foca em um estudo dos pontos fortes e fracos de cada idioma através de análises qualitativas e quantitativas para tentar definir o melhor idioma para determinada situação. A definição do melhor idioma é baseada na modularidade que o mesmo proporciona, não é desejável duplicação, espalhamento e entrelaçamento de código. O resultado alcançado a partir da análise supracitada consiste em um modelo para guiar o desenvolvedor na escolha de determinado idioma. Assim, dada uma *feature* já implementada com AspectJ, a partir de algumas características dos aspectos que implementam a *feature*, é possível indicar um idioma apropriado (no quesito modularidade) para proporcionar flexibilidade de *binding time* daquela *feature*.

Agradecimentos

Agradeço e dedico este trabalho a minha família por todo o apoio em todas as etapas da minha vida. Agradeço aos amigos da faculdade que me apoiaram durante o curso de graduação, em especial a uma pessoa.

Sou extremamente grato a Márcio Ribeiro, por toda a dedicação e paciência comigo, além de me incentivar sempre.

Agradeço também à oportunidade que o professor Paulo Borba me deu de fazer o trabalho de graduação junto com pessoas tão capacitadas e dedicadas.

Índice

1 - Introdução

Linhas de Produto de Software (LPS) [10] são famílias de produtos de software que compartilham um conjunto de artefatos em comum. Elas são desenvolvidas a partir de uma arquitetura em comum, e compartilham alguns artefatos reusáveis. A adoção de LPS é uma solução para desenvolvimento de software que forma sistemas de software que devem adaptar restrições diferentes para satisfazer requisitos dos clientes que possuem expectativas divergentes ou coincidentes [14]. Os sistemas de uma LPS devem ser construídos a partir de uma plataforma. Uma plataforma é um conjunto de artefatos que são comuns a todos os sistemas da LPS, portanto todos os produtos da linha possuem como base essa plataforma [9]. Por conseguinte, não é necessário refazer todo o produto a partir do nada. Para customizá-lo, basta acrescentar os componentes adicionais à plataforma de acordo com os requisitos do cliente. Através do reuso de artefatos em comum entre os diferentes sistemas de uma mesma linha é possível diminuir custos de desenvolvimento, melhorar a qualidade e reduzir o tempo que o software leva até chegar ao mercado [9]. Apesar de todas essas vantagens, o desenvolvimento de LPS implica em investimentos adicionais e mais complexidade, já que a maioria das atividades se propõe a criar artefatos reusáveis [14].

No contexto de LPS é importante definir *feature*. *Features* são incrementos na funcionalidade de um programa, são as unidades semânticas pelas quais programas diferentes dentro de uma família ou LPS podem ser diferenciados e definidos [13]. Através de composições diferentes de *features* vários sistemas podem ser criados dentro de uma LPS. Para implementar *features* em uma LPS, vários mecanismos podem ser utilizados [11, 14, 31], tais como Programação Orientada a Aspectos [32], Padrões de Projeto [5], Arquivos de Configuração, etc.

Outra definição importante é o conceito de *binding time*. *Binding Time* de uma *feature* é o momento em que é decidido a inclusão ou exclusão de uma *feature* em um produto. Nesse contexto, definir uma LPS em que as *features* possuem *binding times* flexíveis é importante [7]. Existem dois tipos de *binding times*: tempo de compilação (estático) e tempo de execução (dinâmico). Eles se aplicam em determinadas situações e condições. Por exemplo, o *binding time* dinâmico deve ser usado para produtos sem restrições de recursos (memória, processamento gráfico, etc), provendo flexibilidade para o usuário selecionar as funcionalidades necessárias sob demanda [7]. No entanto, para ambientes em que há restrições de recursos, o *binding time* estático deve ser escolhido.

No cenário de um jogo de celular em que o usuário determina se deseja ou não jogar com som enquanto o mesmo é carregado, pode ser considerado um exemplo simples de como *binding time* flexível é importante. O fabricante pode fornecer três tipos de jogos: (i) somente com som; (ii) somente sem som; e (iii) usuário escolhe se deseja som ou não. Para celulares que não possuem áudio, o fabricante fornece (ii), para celulares que possuem áudio, mas têm restrições como pouca memória, o fabricante fornece (i) e para celulares avançados com vastos recursos, o fabricante fornece (iii).

Para permitir essa flexibilidade, alguns idiomas [3, 7] baseados em AspectJ foram criados. Neste trabalho, considera-se um idioma como uma técnica que possibilita o desenvolvedor implementar *binding time* flexível para as *features*. No entanto, os idiomas possuem limitações. Ao utilizar o idioma Edicts [7], por exemplo, verifica-se duplicação de código da *feature*. Por conseguinte, este trabalho foca em estudar os pontos fortes e fracos de cada idioma. Para isso, análises qualitativas e quantitativas foram feitas através do uso de métricas de forma a definir qual o melhor idioma para determinada situação. A definição do melhor idioma é baseada na modularidade que o mesmo proporciona, não é desejável duplicação, espalhamento e entrelaçamento de código. Por conseguinte, um guia para o desenvolvedor foi concebido para orientá-lo a escolher o melhor idioma em diversas situações. Por fim, como resultado da análise, este trabalho propõe um modelo para guiar o desenvolvedor a escolher o melhor idioma (no quesito modularidade) para uma determinada situação. Por exemplo, dada uma *feature* já implementada com AspectJ, a partir de algumas características observadas nos aspectos que implementam a *feature*, é possível indicar um idioma apropriado (no quesito modularidade) para proporcionar flexibilidade de *binding time* daquela *feature*.

1.1 – Objetivos

Este trabalho tem os seguintes objetivos:

- Apresentar os idiomas utilizados para implementar *binding time* flexível;
- Avaliar tais idiomas de forma quantitativa e qualitativa;
- Oferecer um modelo para guiar desenvolvedores a escolher o melhor idioma;

1.2 – Organização

O trabalho está organizado da seguinte forma:

- **Capítulo 2:** visa dar toda a fundamentação teórica, para permitir o entendimento do contexto do trabalho: Linha de Produtos de Software, *Binding Time, Features, Feature Model* e Programação Orientada a Aspectos.
- **Capítulo 3:** apresenta os idiomas que serão avaliados: Edicts (3.1), Aspect Inheritance (3.2) e AdviceExecution (3.3).
- **Capítulo 4:** introduz as métricas que serão utilizadas, exibe os resultados das avaliações de acordo com critérios como espalhamento, duplicação e entrelaçamento de código e oferece um modelo para guiar o desenvolvedor.
- **Capítulo 5:** apresentar os pontos estudados, limitações deste trabalho e trabalhos futuros.

2 - Fundamentação teórica

Este capítulo visa introduzir conceitos importantes e fornecer a base teórica necessária para o entendimento deste trabalho. Os conceitos eventualmente são simplificados, porém focados para a compreensão dos capítulos posteriores.

2.1 - Linhas de Produto de Software (LPS)

Linhas de Produto de Software (LPS) englobam uma família de sistemas de software intensivos desenvolvidos a partir de propriedades reusáveis. Com o reuso dessas propriedades, é possível construir uma larga escala de produtos [9]. Outra definição importante diz que uma Linha de Produtos de Software é um conjunto de sistemas intensivos de software que compartilham um conjunto de artefatos que satisfazem as necessidades específicas de um segmento de mercado e que são desenvolvidos a partir de um conjunto de artefatos em comum de forma prescrita [10]. Uma linha de produto é do ponto de vista de mercado, um grupo de produtos semelhantes dentro de um segmento de mercado.

Com o uso de propriedades em comum, um novo produto é formado por componentes de uma plataforma de software. Nesse contexto, plataforma é um conjunto de componentes, subsistemas e interfaces que formam uma estrutura comum a partir da qual vários produtos podem ser desenvolvidos eficientemente [9]. Para desenvolver os sistemas, novos componentes são adicionados à plataforma de maneira que seja criada uma variedade de sistemas resultantes. Com isso, construir um novo sistema se torna mais uma questão de geração do que de criação, portanto a atividade predominante é integração, e não programação.

Linhas de Produto de Software estão emergindo como um novo e importante paradigma de desenvolvimento de software [29]. Linhas de produto simplificam o reuso, pois aproveitam uma grande variedade de propriedades: requisitos, análise do sistema, o sistema completo, arquitetura do software, componentes, documentação, planejamento, testes e dados em geral [10]. Adicionalmente, o conhecimento e habilidade das pessoas envolvidas também são reusáveis. As práticas de construção de conjuntos de sistemas relacionados por propriedades iguais podem render melhoras notáveis na produção, no tempo até chegar ao mercado, na qualidade do produto e no custo de desenvolvimento. Mais especificamente:

- **Redução no custo de desenvolvimento**: com artefatos reusáveis o custo de produção cai significativamente já que vários sistemas são produzidos e não é necessário partir do zero para o desenvolvimento de cada sistema. O gráfico da Figura 2.1.1 exemplifica a redução de custos. Ele compara os custos acumulados em relação ao número de sistemas diferentes dentro de uma LPS. O investimento inicial é maior para uma família de sistema, mas a partir de aproximadamente três produtos, que é o ponto em que as despesas se equivalem, os custos de sistemas únicos crescem mais rápido que os de família de sistema.

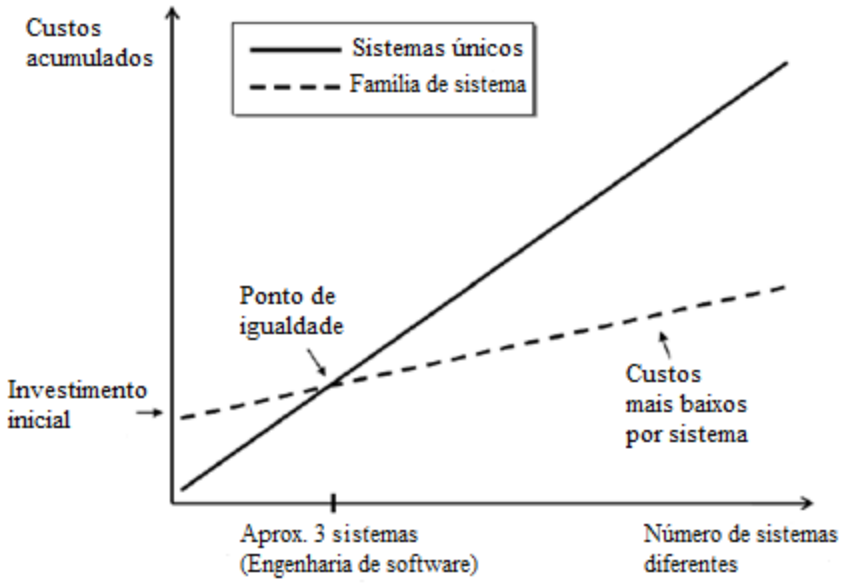

Figura 2.1.1 - Custos para desenvolver **n** tipos de sistemas como sistemas individuais comparado com Linha de Produto de Software [9]

- **Melhora na qualidade**: a partir de propriedades em comum, muitos sistemas diferentes são gerados, por conseguinte, eles são testados e verificados várias vezes e com isso, as chances de encontrar erros e corrigi-los aumentam expressivamente.
- **Redução do tempo até chegar ao mercado**: inicialmente o tempo é longo devido à necessidade de desenvolver uma base para todos os sistemas, mas em seguida, o tempo é reduzido por causa do aproveitamento de componentes previamente construídos. O gráfico da Figura 2.1.2 exibe uma comparação.

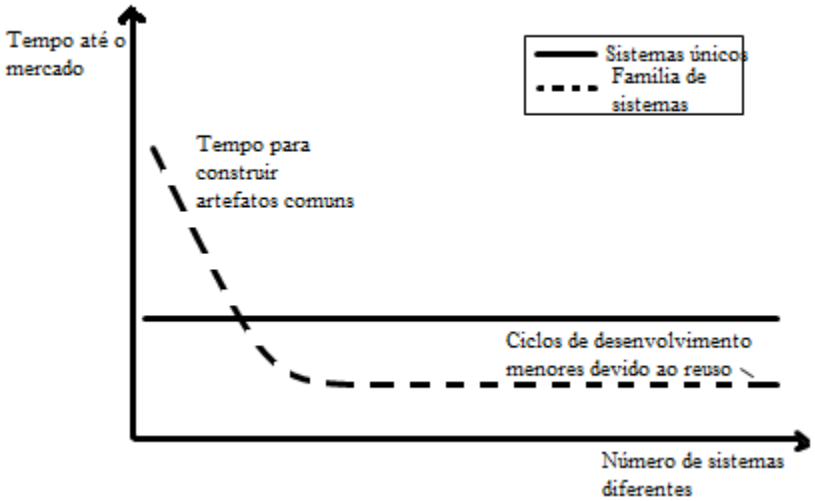

Figura 2.1.2 – Tempo até chegar ao mercado, com e sem linhas de produto [9]

Para obter essas vantagens, é necessário gerenciar cuidadosamente as variações dos sistemas que fazem parte de uma Linha de Produto de Software [12]. É fundamental identificar os pontos em que essas variações se encontram. Um ponto de variação é definido pelas características que diferenciam um sistema de outro dentro de uma mesma Linha de Produto de Software. Essas variações têm papel fundamental nas LPS, já que aplicações diferentes na LPS são individualizadas de acordo com as variações em questão [11].

A Figura 2.1.3 ilustra três carros. A plataforma definida foi reusada para todas as versões do automóvel. No entanto, cada carro tem variações específicas. Os carros B e C não possuem frisos cromados, faróis de milha e antena para rádio, já o carro A possui. Os carros A e C possuem rodas de liga leve enquanto que o carro B não possui. Os três automóveis possuem o mesmo design e compartilham diversas características em comum. As variações entre os três modelos influenciam no preço do produto e permitem a customização em massa e conseqüentemente possibilita uma abrangência entre usuários com diferentes condições financeira e objetivos. Esse exemplo demonstra como as variações têm papel fundamental nas LPS.

Carro A

Carro B

Carro C

Figura 2.1.3 – três carros de um mesmo modelo, mas com suas variações particulares

2. 2 - Feature e Feature Model

Features são incrementos na funcionalidade de um programa. Elas são as unidades semânticas através das quais programas diferentes em uma mesma LPS podem ser diferenciados e definidos [13]. As *features* definem os aspectos comuns entre membros de uma LPS e também

diferenças entre esses membros. No exemplo da seção anterior, uma *feature* comum entre os automóveis é o farol e uma diferente é o friso cromado do carro A da Figura 2.1.3.

Composições diversas de *features* formam diferentes programas dentro de uma LPS. *Feature Models* esboçam essas composições de forma simples, mas completa, ou seja, exibindo todas as variações possíveis que os sistemas podem ter dentro de uma LPS. Um *feature model* consiste basicamente de diagramas que representam as *features* e seus relacionamentos [15]. Existem quatro tipos de *features* no modelo introduzido:

- **Obrigatória:** a *feature* tem que estar presente sempre;
- **Opcional:** a *feature* pode estar inclusa ou não;
- **Alternativa (XOR):** *features* desse tipo pertencem a um grupo no qual somente uma pode ser selecionada;
- **Ou (OR):** *features* desse tipo pertencem a um grupo no qual mais de uma *feature* pode ser selecionada.

O desenho da composição das *features* descreve uma estrutura hierárquica com relacionamentos obrigatórios, opcionais e alternativos. *Features* são representadas por retângulos que contém seus nomes e os relacionamentos entre eles são representados por linhas e arcos [14].

Na Figura 2.2.1, há um exemplo de um *feature model* simples, mas que contém todos os tipos de relacionamentos. O esquema ilustra uma *feature* e-shop de uma família de sistemas de compra online e suas possíveis configurações. As *features Catalogue, Payment e Security* são obrigatórias. O pagamento (*Payment*) pode ser de dois tipos: transferência bancária (*Bank transfer*) ou cartão de crédito (*Credit card*), portanto elas são *features* Ou (OR). A segurança (*Security*) só pode ser feita de duas maneiras ou com segurança alta (*High*) ou com segurança padrão (*Standard*), jamais pode ser feita dessas duas maneiras ao mesmo tempo, o que caracteriza uma *feature* Alternativa. A busca (*Search*) pode estar inclusa ou não, exemplificando uma *feature* Opcional.

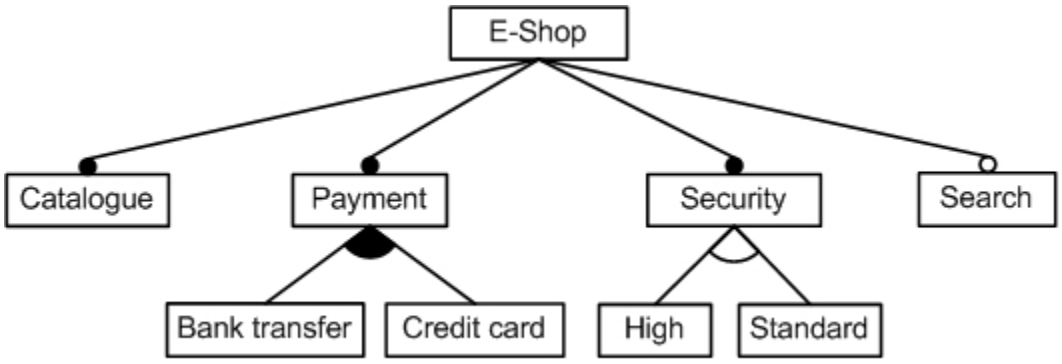

Figura 2.2.1 – *Feature Model* de um e-shop

Esse modelo é amplamente utilizado [30], pois facilita bastante a visualização de todas as variações possíveis de produtos de uma LPS.

2.3 - Binding Time

A seleção de *features* em um produto pode ocorrer em momentos distintos. Por exemplo, um jogo desenvolvido para vários tipos de celulares possui uma *feature Screen Resolution* que define a resolução da tela. Quando o celular suporta apenas uma resolução de tela, somente há uma possibilidade de configuração, portanto a resolução pode ser selecionada previamente – ex., durante a compilação do software. No entanto, vários dispositivos móveis suportam mais de uma resolução de tela – ex., celulares *slide*, em que a tela se ajusta à posição – e precisam ser configuráveis posteriormente – ex., em tempo de execução. Na Figura 2.3.1 é exibido um modelo de celular que suporta apenas uma resolução de tela. Na Figura 2.3.2 é exibido um modelo de celular que suporta pelo menos duas resoluções diferentes enquanto o usuário utiliza-o.

Figura 2.3.1 – Celular que suporta apenas uma resolução de tela

Figura 2.3.2 – Celular com *slide* que suporta mais de uma resolução de tela

Outra situação prática em que *features* são selecionadas em momentos diferentes: um termômetro de residências brasileiras deve usar a escala Celsius. Para residências americanas o ideal é a escala Fahrenheit. Em ambos os casos, a *feature* escala é selecionada previamente (estaticamente). Não há como alterá-la. No entanto, esse termômetro deve fornecer as duas escalas para hotéis, pois hóspedes de diferentes regiões precisam usá-lo [3]. Nessa situação, o hóspede seleciona a escala posteriormente (dinamicamente).

Para incrementar o número de produtos na linha, uma empresa pode desejar desenvolver produtos para as situações anteriores. Nesse caso, deseja-se criar dois softwares que possuam *features* em comum, mas que se diferenciam em quando a *feature* é selecionada.

Na terminologia de Linha de Produto de Software, pontos de variação controlam a inclusão ou exclusão de *features* de um produto que faz parte de uma linha de produto [7]. *Binding time* refere-se ao momento em que decisões para um ponto de variação são configuradas [2]. Em outras palavras, o *binding time* de uma *feature* é o momento em que ocorre a decisão se a *feature* vai ser incluída ou excluída do produto.

Existem diferentes tipos de *binding times* [7], os principais são:

- **Pré-processamento e tempo de compilação** – a *feature* pode ser incluída antes da compilação, usando recursos como compilação condicional [33]. Outra forma é a *feature* ser incluída na geração do *build*, em tempo de compilação.
- **Tempo de ligação** – a ligação pode ser executada várias vezes. Pode acontecer imediatamente após a compilação ou até depois de o programa começar a executar.
- **Tempo de inicialização** – é uma fase prévia ao começo da execução do programa. A *feature* é incluída, por exemplo, analisando propriedades de um arquivo na inicialização de um programa.
- **Tempo de execução** – a seleção da *feature* ocorre enquanto o programa é executado, por exemplo, o usuário pode escolher entre visualizar ou não a *feature*.

Features podem ser incluídas ou excluídas de forma estática (tempo de compilação ou em um passo de pré-processamento) ou de forma dinâmica (quando o programa é carregado ou em tempo de execução). Ambos possuem vantagens e desvantagens: estaticamente, facilita a customização sem custo no tempo de execução, mas uma vez que a *feature* é incluída, não é possível desabilitá-la; dinamicamente, permite que sejam habilitadas *features* em tempo de execução, no entanto há um custo adicional de consumo de memória e queda de desempenho. Este trabalho foca nesses dois tipos de *binding time*: **estático** e **dinâmico**.

No contexto de *binding time* dinâmico, é importante definir os critérios que determinam se uma *feature* deve ou não ser incluída no sistema. Um *driver* é um mecanismo (ou um conjunto de mecanismos) responsável por fornecer informação para decidir se uma *feature* deve ser habilitada ou não em tempo de execução [3]. Esses mecanismos variam de uma simples interface gráfica que pergunta se o usuário quer que seja habilitada, até formas mais complexas como sensores que decidem baseados em determinados fatores [3], como, por exemplo, sensores de fumaça e temperatura que habilitam uma *feature* para chamar os bombeiros em uma casa inteligente.

2.4 - Programação Orientada a Aspectos (POA)

Programação Orientada a Aspectos é um paradigma que permite aos desenvolvedores de software separar e organizar o código de acordo com a sua importância para a aplicação (*separation of concerns*). POA procura solucionar a ineficiência em capturar algumas das importantes decisões de projeto que um sistema deve implementar. Essa dificuldade faz com que a implementação dessas decisões de projeto seja distribuída pelo código, resultando num entrelaçamento e espalhamento de código com diferentes propósitos. Esse entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção desses sistemas difícil. Dessa forma, POA aumenta a modularidade separando código que implementa funções específicas, afetando diferentes partes do sistema, chamadas preocupações ortogonais (*crosscutting concern*). Exemplos de *crosscutting concerns* são persistência, distribuição, controle de concorrência, tratamento de exceções, e depuração [16].

Uma linguagem comumente usada em POA é AspectJ [17]. A principal estrutura de AspectJ é um aspecto. Com um aspecto é possível alterar e adicionar membros (método, atributo e construtor) de classes, além de alterar a hierarquia do sistema e manipulação de exceções. Também é possível alterar ou adicionar comportamento de um programa através da interceptação de pontos no fluxo de execução, chamados de *join points.* Comumente são definidas nos aspectos estruturas chamadas *pointcuts*, que são a descrição lógica de um conjunto de *join points.* É necessário definir qual comportamento o aspecto deve apresentar diante de um conjunto de *join point*s, para isso há uma estrutura chamada *advice.* Um *advice* além de encapsular o comportamento, determina se o mesmo vai ser executado antes (*before*), depois (*after*) ou em volta (*around*), esse último, a propósito, pode manipular o retorno de métodos. Outro recurso muito utilizado são os *intertypes*. *Intertypes* permitem a alteração estática da estrutura de um programa: membros de classes (método, atributo e construtor) podem ser adicionados e a estrutura hierárquica pode ser modificada. *Intertypes* sempre operam em tempo de compilação. A figura 2.4.1 mostra um exemplo da estrutura de um aspecto para controle de transação.

```
public aspect AspectoControleTransacao {

	// Intertype é usado para adicionar o atributo conexao
	// à classe Fachada.
	private Connection Fachada.conexao;

	// Pointcut definido para capturar a execução de todos os
	// métodos da classe Fachada, exceto o método obterInstancia()
	pointcut controleTransacao() : execution(* Fachada.*(..))
		&& !execution(* Fachada.obterInstancia());

	// Advice definido para criar uma transação antes da
	// execução dos métodos capturados pelo pointcut acima
	before() : controleTransacao() {
		conexao.createTransaction();
	}

	// Advice definido para efetivar a transação depois
	// da execução dos métodos capturados pelo pointcut acima,
	// caso não tenha ocorrido nenhum erro
	after() returning() : controleTransacao() {
		conexao.commitTransaction();
	}

	// Advice definido para desfazer as alterações feitas
	// durante a transação depois da execução dos métodos
	// capturados pelo pointcut acima, caso tenha ocorrido
	// algum erro
	after() throwing() : controleTransacao() {
		conexao.rollbackTransaction();
	}
}
```

Figura 2.4.1 – Exemplo da estrutura de um aspecto para controle de transação

A Programação Orientada a Aspectos provê vários benefícios e recursos para deixar os programas mais modulares. Desta forma, o reuso de componentes é facilitado, além de reduzir custos com manutenção e evolução.

3 - Idiomas para implementar *Binding times* flexíveis em LPS

Definir uma linha de produtos em que *features* têm *binding times* diferentes é importante [4]. Algumas linhas de produto de software precisam de pontos de variação com *binding time* flexível. Produtos com recursos limitados, por exemplo, devem usar *binding time* estático devido ao custo de desempenho introduzido pelo dinâmico.

Para permitir o desenvolvimento de *features* com *binding times* flexíveis em linhas de produtos de software, alguns idiomas foram propostos. Esses idiomas são baseados em AspectJ e fornecem a possibilidade de mudar o *binding time* de uma *feature* dependendo da combinação de aspectos que é colocada no momento da compilação do produto. Os idiomas avaliados neste trabalho são: Edicts [7], Aspect Inheritance [3] e AdviceExecution [3]. Este capítulo fornece uma breve apresentação dos mesmos, ao passo que o Capítulo 4 faz a avaliação de tais idiomas com respeito à modularidade das *features*.

Neste capítulo, o jogo Tetris J2ME será usado como exemplo. A *feature* analisada se chama *NextPieceBox*. Ela é responsável por mostrar a próxima peça que deve aparecer na tela do jogo. Esta *feature* é opcional, portanto é possível criar jogos com e sem a exibição da próxima peça. É importante salientar que três versões do produto podem ser geradas com a variação do *binding time* da *feature*: (i) sem exibição da próxima peça (*binding time* estático); (ii) com exibição da próxima peça sem a possibilidade de desabilitá-la (*binding time* estático); (iii) com exibição da próxima peça e com a possibilidade de desabilitá-la (*binding time* dinâmico) [3]. A Figura 3.1 exibe dois tipos de produtos do Tetris.

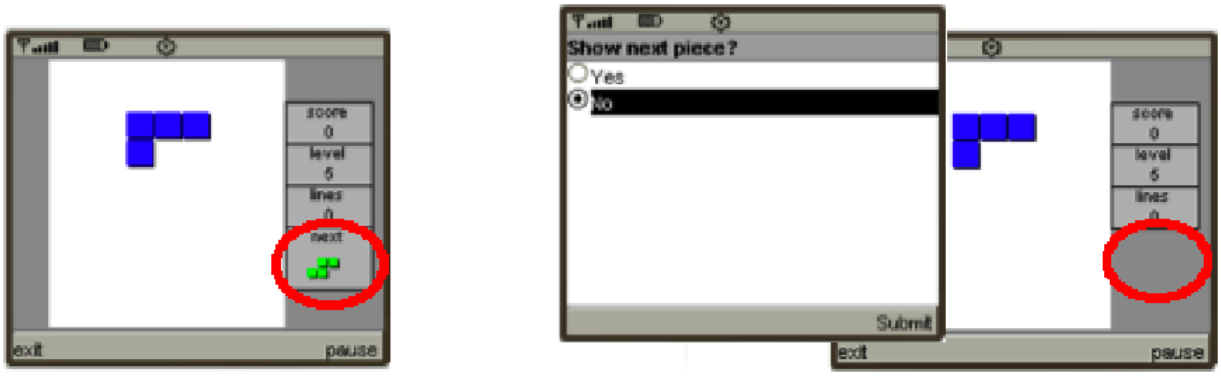

Figura 3.1 – Do lado esquerdo, *NextPieceBox* estático, do lado direito dinâmico

As seções seguintes apresentam os idiomas propostos.

3.1 - Edicts

Edicts é uma técnica para implementar flexibilidade de *binding time* de *features* de forma modular e conveniente [7]. Edicts usa programação orientada a aspectos em combinação com

padrões de projeto para obter o *binding time* flexível. Primeiramente, o ponto de variação é encapsulado dentro da implementação de um padrão de projeto [5]. O padrão serve para deixar o ponto de variação identificável e estável, e fornecer os mecanismos necessários para alterar o *binding time.* Por conseguinte, o Edict é codificado para manipular o *binding time* dos pontos. Um Edict é um aspecto que implementa uma estratégia para seleção de uma *feature* [7]. Para aplicar o Edict, é fundamental identificar o ponto de variação encapsulado no padrão de projeto.

Padrões de projeto são uma técnica comum e popular para descrever coleções de elementos de um programa que funcionam de forma conjunta [5]. Padrões são úteis, pois eles auxiliam as pessoas a entender a codificação do software através de implementações estruturadas. Eles ajudam a descrever tanto o propósito quanto a modularidade esperada de um conjunto de elementos de um programa. Padrões de projeto permitem que os relacionamentos entre artefatos do programa sejam mais evidentes e ajudam a criar e manter pontos de estabilidade na arquitetura do software [7]. A expressividade e estabilidade dos padrões de projeto o tornam adequado para implementar muitos tipos de conexões de *features*. Além disso, eles são adequados para implementar pontos de variação.

Implementações típicas de padrões de projeto visam dar suporte a decisões em tempo de execução. No entanto, foi demonstrado em trabalhos prévios que padrões de projeto também podem ser úteis para implementar decisões que são tomadas antes do tempo de execução, quando o sistema é compilado ou inicializado [6].

Apesar de ser possível implementar relacionamentos estáticos através de mecanismos dinâmicos, essas técnicas tendem a esconder informações que podem ser usadas estaticamente para verificação e otimização do software [7]. Por conseguinte, quando os relacionamentos de um padrão são conhecidos estaticamente, é desejável que seja escolhida uma implementação que forneça legibilidade para facilitar o desenvolvimento e a análise.

Programação Orientada a Aspectos fornece os mecanismos necessários para que em combinação com os padrões de projeto seja possível definir os *binding times* [7]. Os *pointcuts* e *advices* modularizam a manipulação às classes que implementam os padrões de projeto. É possível definir vários aspectos para manusear um mesmo ponto de variação – ex., um aspecto para *binding time* estático e outro para dinâmico. Esses aspectos definidos são chamados de Edict.

Edict é codificado com AspectJ, um *template* pode ser visto na Figura 3.1.1 [7].

```
public aspect Edict {
        // Usar interfaces para definir os papéis
        // das classes na implementação do padrão de projeto
        public interface [Role] {};
        declare parents: [Class] implements [Role];

        // Usar intertypes para implementar os métodos do padrão
        // para o binding time desejável
        ... [Role].[method](...) { ... }

        // Usar pointcuts para identificar interações
        // entre as classes do padrão de projeto
        pointcut [pcname](...):
        [call or execution](...)
        [&& target(...)] [&& within(...)] && args(...);

        // Usar advice around para implementar o binding time
        // nos pontos de interação
        ... around(...): [pcname](...) { ... }
}
```

Figura 3.1.1 – *template* de um Edict

O uso de *intertype* e *around advices* permite o controle sobre os pontos de variação necessários para definir um *binding time* em particular. Um Edict pode conter quantos *advices* e *intertypes* forem necessários para controlar o ponto de variação em questão. Com a implementação dos pontos de variação através de padrões de projeto, fica simplificada a identificação dos *join points* pelos *pointcuts* de AspectJ. Normalmente, apenas *join points* como *call* e *execution* são necessários para identificar os pontos de variação e eventualmente, *within* e *target* podem ser usados para restringir a abrangência do *advice*.

Edicts tem a intenção de obter uma otimização estática, por isso é aplicado quando o sistema é compilado [7]. O desenvolvedor tem o papel de escolher o Edict responsável pelo *binding time* desejado e o inclui no *build.*

Uma maneira recomendada da arquitetura de Edicts é exibida na Figura 3.1.2 [7]. Um aspecto abstrato possui a declaração dos *pointcuts* necessários para controlar os pontos de variação da *feature*, enquanto que dois aspectos concretos estendem do abstrato e implementam os *advices*, um aspecto para o *binding time* dinâmico (*LateBinding*) e outro para o estático (*EarlyBinding*).

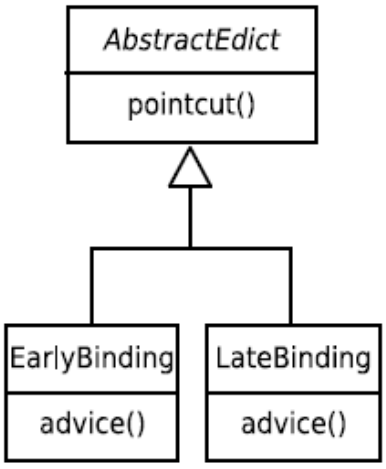

Figura 3.1.2 – Arquitetura da implementação de um Edict

Existe um conjunto de passos definido por V. Chakravarthy [7] que indica um método de utilizar Edicts para introduzir *binding time* flexível.

1. **Identificar e caracterizar os pontos de variação**
   Nesse passo é necessário analisar o comportamento do software e identificar os pontos de variação com ajuda de um *feature model*. Após a identificação, o programador escolhe os padrões de projetos mais adequados.
2. **Expressar *concerns* de domínio usando *binding time***
   Os pontos de variação são examinados assim como os padrões de projeto escolhidos. O padrão de projeto escolhido deve ser flexível o suficiente para que o *assembler* possa fazer a escolha entre adicionar ou remover variações e adicionar comportamento estático ou dinâmico.
3. **Implementar o padrão de projeto**
   Uma resolução é proposta por E. Gamma [5] em que os padrões são implementados no código participante. Outra abordagem é descrita por J. Hannemann [8] em que os papéis são encapsulados e abstrai o padrão da aplicação.
4. **Introduzir flexibilidade de *binding time* usando Edicts**
   O programador codifica *pointcuts* para interceptar as decisões de *binding time* dinâmico em um aspecto abstrato. O padrão de projeto original é refatorado para remover o código para o *LateBinding* Edict. Similarmente o *EarlyBinding* Edict é codificado.
5. **Otimizar os produtos resultantes**
   Edicts dá assistência a "otimizadores" de código estimulando informação estática em casos envolvendo *binding times* estáticos.

Usando a abordagem de Edicts para a *feature NextPieceBox* do Jogo Tetris, introduzido anteriormente, dois aspectos vão ser implementados, um para *binding time* estático e outro para o dinâmico, este último contém a implementação do *driver*, que nesse contexto consiste da entrada do usuário na tela do jogo. O usuário decide se deseja que a *feature NextPieceBox* seja habilitada ou não.

A *feature NextPieceBox* é composta pela classe *NextPieceBox* e três fragmentos de código na classe TetrisCanvas. De acordo com o idioma Edicts, esses fragmentos devem ser extraídos para aspectos, portanto se a *feature* em questão estiver desabilitada, não devem estar incluídos no *build* nem os fragmentos de código nem a classe *NextPieceBox*.

Na Figura 3.1.3 é exibida a implementação do *DynamicNextPieceBox* [3].

```
1  public privileged aspect DynamicNextPieceBox {
2    private NextPieceBox nextPieceBox;
3
4    after(TetrisCanvas canvas): nextPiece(canvas) {
5      if (getUserChoice()) {
6        nextPieceBox = new NextPieceBox(...);
7      }
8    }
9
10   after(Graphics g, TetrisCanvas canvas):
11       infoBoxes(g, canvas) {
12     if (getUserChoice()) {
13       ... nextPieceBox.setPieceType(
14         canvas.game.getNextPieceType()); ...
15     }
16   }
17
18   after(Graphics g): paintOnce(g) {
19     if (getUserChoice()) {
20       ... nextPieceBox.paint(g); ...
21     }
22   }
23
24   before(Command command, TetrisMidlet midlet):
25       commandAction(command, midlet) {
26     if (command == submitUserChoice) {
27       ... userChoice = nextPieceList.getString(
28         nextPieceList.getSelectedIndex()); ...
29     }
30   }
31
32   private boolean getUserChoice() {
33     return userChoice.equals("Yes");
34   }
35 }
```

Figura 3.1.3 – Implementação do aspecto *DynamicNextPieceBox*

O aspecto consiste não só do código referente à *feature NextPieceBox* (linhas 4 a 22), mas também da implementação do *driver* (linhas 24 a 34). O *driver*, nesse caso, é baseado em uma resposta do usuário na inicialização do jogo. Uma tela pergunta se o usuário deseja habilitar a *feature*, caso a resposta seja positiva a mesma é incluída e o código encapsulado é executado, caso contrário o código não é executado.

De acordo com a técnica Edicts, outro aspecto tem que ser criado para ser usado com *binding time* estático. O *StaticNextPieceBox* é diferenciado do *DynamicNextPieceBox* pois não possui a implementação do *driver*, já que esse não faz sentido em um cenário estático. No entanto, o código referente à *feature NextPieceBox* está duplicado em ambos os aspectos. A duplicação e espalhamento de código acarretam em maior dificuldade para manutenção e torna o sistema mais propício a erros. O espalhamento do *driver* também é crítico, pois existe pelo menos uma cláusula condicional por *advice* dentro do aspecto de Edicts. Diversos problemas podem surgir com essa situação, se alguma cláusula for esquecida, podem ocorrer exceções em tempo de execução, por exemplo. Esse problema pode se tornar pior caso vários *drivers* façam parte da implementação, pois caso esse *driver* seja independente do que já está integrado, outro aspecto deve ser criado.

3.2 - Aspect Inheritance

Com o intuito de eliminar a duplicação e espalhamento de código introduzido por Edicts, outro idioma foi desenvolvido, desta vez baseado em herança de aspectos e definições de *pointcuts* abstratos [3].

Aspect Inheritance é um idioma para implementar *binding time* flexível. Aspect Inheritance é baseado em conceitos de herança de aspectos e definição de *pointcuts* abstratos. Primeiramente o ponto de variação é identificado e seu código deve ser refatorado para um aspecto. O aspecto deve ser abstrato e contém todo o código referente ao comportamento da *feature*; *pointcuts*, *advices* e *intertypes* normalmente são suficientes. O aspecto abstrato é útil para evitar a duplicação de código mencionada no idioma Edicts e dar suporte à implementação do *driver*. Em seguida, um *pointcut* abstrato para cada *driver*, que possa existir, é definido e associado aos *pointcuts* já existentes. Posteriormente, é necessário codificar dois aspectos, um para *binding time* estático e o outro para *binding time* dinâmico. Essa estrutura pode ser visualizada na Figura 3.2.1.

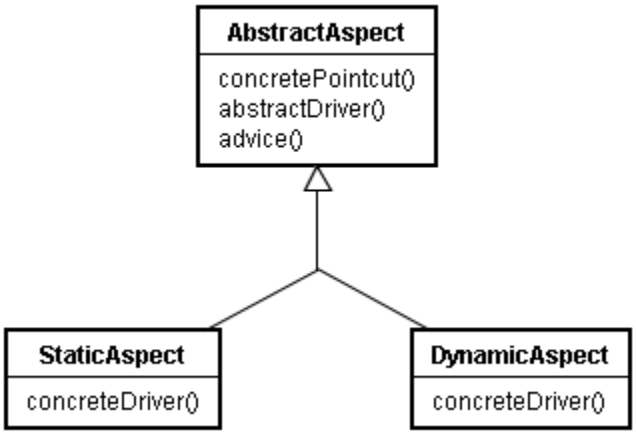

Figura 3.2.1 – Arquitetura da implementação de um Aspect Inheritance

A diferença básica entre Edicts e Aspect Inheritance é que no primeiro o código da *feature* é definido nos aspectos concretos, o que acaba acarretando na duplicação. Aspect Inheritance mantém todo o código da *feature* no aspecto abstrato evitando a duplicação nos aspectos concretos, que somente implementam o código dos *drivers.*

A Figura 3.2.2 [3] mostra o aspecto abstrato *NextPieceBox*. Esse aspecto é responsável por implementar a *feature NextPieceBox* e pela definição do *pointcut* abstrato (Linha 2). O *pointcut* do *driver* fica associado aos já existentes (Linhas 4, 7 e 10) para que sejam habilitados de acordo com o *pointcut* do *driver*. Novamente, há problema de espalhamento de código (associação dos *drivers*).

```
1  public privileged abstract aspect NextPieceBox {
2    abstract pointcut driver();
3
4    pointcut nextPiece(TetrisCanvas canvas):
5        ... && driver();
6
7    pointcut infoBoxes(Graphics g,
8        TetrisCanvas canvas): ... && driver();
9
10   pointcut paintOnce(Graphics g,
11       TetrisCanvas canvas): ... && driver();
12 }
```

Figura 3.2.2 – Definição do aspecto abstrato

Dois aspectos são definidos para implementar os *drivers.* A Figura 3.2.3 [3] mostra dois aspectos responsáveis por essa tarefa. O aspecto *DynamicNextPieceBox* implementa o código do *driver* (o mesmo mecanismo da tela citado em Edicts) definindo o *pointcut* concreto que retorna *true* se o usuário desejar habilitar a *feature*. O aspecto *StaticNextPieceBox* não precisa implementar nenhum *driver*, ele é definido para ser sempre *true* [3].

```
1  public privileged aspect DynamicNextPieceBox
2      extends NextPieceBox {
3
4    pointcut driver(): if (userChoice.equals("Yes"));
5
6    // Driver code...
7  }
8  public privileged aspect StaticNextPieceBox
9      extends NextPieceBox {
10
11   pointcut driver() : if (true);
12 }
```

Figura 3.2.3 – Definição dos aspectos concretos

O problema mais grave (duplicação de código) foi resolvido. E a integração de outros *drivers* foi facilitada, pois basta estender o aspecto *NextPieceBox* e definir um *pointcut* concreto. No entanto, o aspecto *StaticNextPieceBox* existe para implementar algo que não faz sentido, *binding time* estático não necessita de *driver*. O espalhamento de código visto na Figura 3.2.2, dificulta a manutenção de código, pois se algum "*&& driver*" for esquecido, o sistema fica vulnerável a exceções em tempo de execução. Para casos em que a *feature* é mais complexa e requer mais de um aspecto para sua modularização, outro problema foi identificado: os aspectos concretos têm que ser implementados para cada aspecto abstrato, já que AspectJ não permite herança múltipla. Com isso, muitos aspectos devem ser criados em casos que existam vários *drivers* diferentes.

3.3 - AdviceExecution

Os idiomas mostrados apresentam problemas como duplicação, espalhamento de código e dificuldades para adicionar novos *drivers*. Visando resolver os problemas citados, outro idioma foi proposto [3].

AdviceExecution é um idioma baseado em *pointcuts adviceexecution* pra implementar flexibilidade de *binding times* de *features*. Inicialmente, os pontos de variação são identificados e refatorados para aspectos, dependendo da complexidade da *feature*, podem ser criados quantos aspectos forem necessários. Após definidos os aspectos que encapsulam os pontos de variação da *feature*, é necessário criar um aspecto que contenha o *pointcut adviceexecution*. Esse *pointcut* intercepta a execução de *advices* dos outros aspectos definidos anteriormente e, baseado na condição do *driver*, executa ou não os *advices.* A restrição de *advices* que o *pointcut adviceexecution* abrange pode ser definida utilizando *within* e *!within*. Por uma questão de simplificação, é aconselhável que todos os aspectos referentes a uma *feature* estejam no mesmo pacote. Isso ajuda na definição do *pointcut*, como será mostrado. A Figura 3.3.1 demonstra um *template* de um *pointcut adviceexecution.*

```
1  Object around() : adviceexecution()
2        && within(algum.pacote.*)
3              && !within(AdvExecAspect) {
4
5        if (userChoice.equals("Yes")) {
6              return proceed();
7        }
8        return null;
9  }
10 // Código do driver
```

Figura 3.3.1 – *Template* de um *pointcut adviceexecution*

Este idioma não apresenta duplicação de código e a implementação do *driver* não está espalhada: toda a verificação do *driver* está encapsulada dentro de um *pointcut* como o da Figura 3.3.1 e é compatível com todos os aspectos de determinado pacote. Adicionar novos *drivers* não é problemático, pois basta definir outro aspecto que encapsule o código e restringi-lo aos aspectos que o mesmo deve abranger.

AdviceExecution aplica-se perfeitamente quando a *feature* utiliza *advices before, after e around*. No entanto, pode haver uma *feature* que precise de um *advice* a*round* que retorne algum objeto. Nesse cenário, quando o usuário decidir desabilitar a *feature*, o *advice* em questão retorna *null*, como definido na Linha 8 da Figura 3.3.1. Essa situação causa uma exceção *NullPointerException* no código base. Não é possível alterar de *Object* para *void*, pois algumas variações devem ser encapsuladas por *around advices* que retornam algum tipo de objeto e não compilam com o uso de *void.*

Várias implementações foram feitas com o uso dos três idiomas apresentados neste capítulo fazendo uso de aplicações reais e com um nível de complexidade mais alto do que a *feature* do jogo Tetris. Os resultados são registrados no próximo capítulo.

4 - Avaliação

Neste capítulo, os idiomas propostos no capítulo anterior são avaliados de acordo com a modularidade que eles proporcionam. Mais especificamente, este trabalho trata dos seguintes critérios: espalhamento, duplicação e entrelaçamento de código. Para essa avaliação, são utilizadas métricas. Por isso, elas são explicadas e aplicadas em sistemas reais. Portanto, os sistemas reais utilizados para a avaliação são apresentados, bem como suas *features* envolvidas no estudo. Os resultados são mostrados para cada *feature* de cada estudo de caso. Por fim, um modelo é apresentado para guiar desenvolvedores a escolherem o melhor idioma para cada situação.

A abordagem usada para as métricas de *software* foi GQM (Goal, Question, Metric) [34]. GQM é um modelo de medida baseado em três níveis:

- **Nível conceitual (goal):** identificar os objetivos a serem alcançados no estudo;
- **Nível operacional (question):** questões são usadas para definir um modelo de avaliação para ajudar a chegar aos objetivos;
- **Nível quantitativo (metric):** um conjunto de métricas, baseadas nos modelos, é associado a cada questão de forma a respondê-las.

4.1 - Objetivo

O principal objetivo consiste em avaliar idiomas que implementam *features* com *binding time* flexível. Essa avaliação é baseada em modularidade. Os principais interesses são: duplicação de código, espalhamento e entrelaçamento de código da *feature* e do *driver*.

4.2 - Questões

Para alcançar o objetivo do trabalho, as seguintes questões devem ser respondidas:

- **Q1:** Qual idioma reduz duplicação de código quando implementa flexibilidade de *binding time*?
- **Q2:** Qual idioma reduz espalhamento dos códigos das *features* e dos *drivers*?
- **Q3:** Qual idioma reduz entrelaçamento entre os códigos das *features* e dos *drivers*?
- **Q4:** Qual idioma precisa de menos linhas de código e de números de componentes?

4.3 - Métricas

Para responder as questões apresentadas, foram avaliadas implementações diferentes de sistemas reais através de métricas que são amplamente utilizadas [18, 19, 20, 21]. Portanto, as seguintes métricas são levadas em conta:

- ***Pairs of Clone Code* (PCC):** contagem do número de pares de clone de código resultantes da modularização dos *drivers* e *features* implementada por um idioma;

- ***Degree of Scattering* (DOS) e *Concern Difusion over Components* (CDC):** quantifica quão dispersa está a implementação das *features* avaliadas e dos *drivers*;
- ***Degree of Tangling* (DOT):** mede o quão dedicado um modulo está referente às *features* avaliadas;
- ***Source Lines of Code* (SLOC):** é a contagem do número de linhas de código fonte necessárias para implementar cada *feature* avaliada;
- ***Vocabulary Size* (*VS*):** contagem do número de componentes (classes, aspectos e arquivos de configuração).

O *Degree of Scattering* de uma *feature* é calculado pela normalização de sua concentração referente a cada unidade modular $m \in$ M (o conjunto de unidades modulares de um estudo de caso) [18]. Essa métrica admite valores entre zero (uma *feature* que é completamente localizada em um único módulo) e um (uma *feature* que é igualmente espalhada pelas unidades modulares). Semelhantemente, o *Degree of Tangling* (DOT) de um módulo é calculado pela normalização de sua dedicação referente a cada *feature* $f \in$ F (o conjunto de *features*) [18]. Além disso, DOT admite valores entre zero (um módulo totalmente fora de foco) e um (um módulo completamente dedicado a uma *feature*). Classes e aspectos são considerados as unidades modulares da avaliação. Para diferenciar implementações que resultam em DOS similares, a métrica CDC também é considerada [21]. DOS e CDC são usadas principalmente porque alguns idiomas têm DOS de implementações de *features* similares, apesar de serem significativamente diferentes de acordo com o número de componentes necessários para implementar a mesma *feature*. Assim sendo, as métricas DOS e CDC se complementam.

Source Lines of Code (SLOC) e *Vocabulary Size* (VS) são métricas conhecidas para quantificar o tamanho e complexidade de um módulo. *Pairs of Code* (PCC) podem indicar uma implementação que é capaz de causar danos à manutenção [22, 23, 24], já que alterações em determinado código clonado têm que ser propagadas para todos os clones. Apesar de haver discussões que propõem que alguns padrões de clone não são considerados danosos ao sistema [24], os clones encontrados neste trabalho podem determinar se um idioma é ruim, como por exemplo, podem dificultar a tarefa de manutenção do código das *features*.

A Tabela 4.3.1 faz o relacionamento entre as questões e métricas:

Questão	Métrica
Q1	PCC
Q2	DOS e CDC
Q3	DOT
Q4	SLOC e VS

Tabela 4.3.1 – relacionamento entre questões e métricas

4.4 – Estudos de caso

Nesta seção são apresentados os sistemas reais que tiveram as *features* modularizadas e implementadas com *binding times* estáticos e dinâmicos.

4.4.1 – Freemind

Freemind [25] é um sistema *open-source* escrito em Java para a elaboração de *mind maps*. *Mind maps* permitem a criação de uma estrutura hierárquica de um conjunto de idéias sobre um conceito, portanto são muito úteis para a atividade de *brainstorming* e para o desenvolvimento de projetos em geral.

A Figura 4.1.1.1 mostra um exemplo de um *mind map.* A idéia chave desse exemplo é organizar a construção de um trabalho de graduação, definir prazos e metas. Para auxiliar a visualização, o Freemind disponibiliza alguns recursos. As pequenas imagens são chamadas de *icons* e o contorno em volta do nó "Capítulo 4" é chamado de *cloud*. Esses dois recursos representam as *features* que foram modularizadas. Ambas são *features* opcionais dentro do *feature model*.

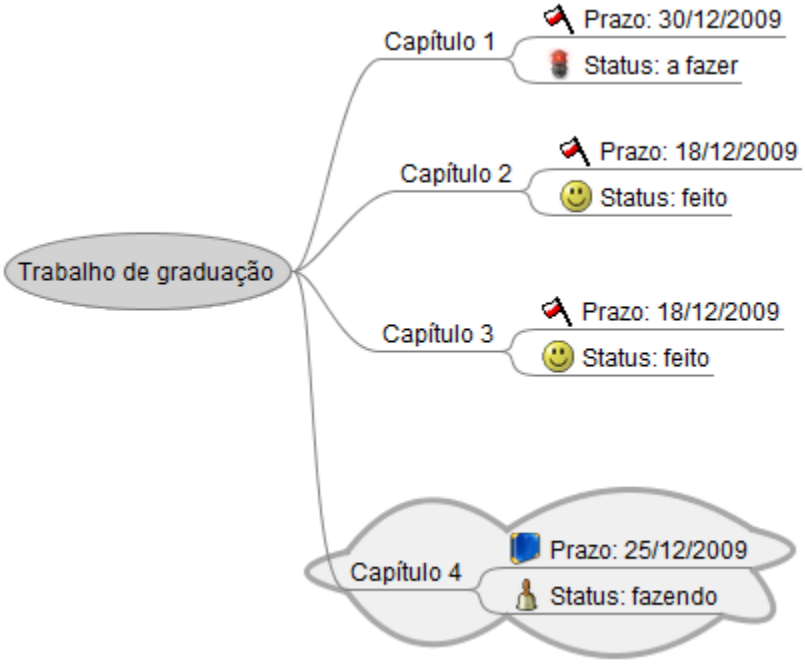

Figura 4.1.1.1 – Exemplo de *mind map*

As duas *features Icons* e *Clouds* estão espalhadas e entrelaçadas entre as camadas da arquitetura do sistema. Após a extração das *features* é possível gerar produtos em que o usuário escolhe qual das duas *features* deseja habilitar dinamicamente. Portanto, todas as combinações de estático a dinâmico, de habilitado a desabilitado podem ser geradas. Alguns exemplos da modularização das *features* e implementação do *binding time* flexível com os três idiomas podem ser encontrados no apêndice A.

4.4.2 - ArgoUML

ArgoUML é uma ferramenta open-source para modelagem UML que inclui suporte para todos os padrões de diagramas UML 1.4 [26]. A ferramenta permite que o usuário customize alguns recursos como, por exemplo, alterar a maneira como as notações são exibidas. A notação UML 1.4 permite os seguintes símbolos: "+", "-" e "#" enquanto que a notação Java permite:

“public”, “private” e “protected”. Esse recurso do ArgoUML representa uma *feature* que foi extraída e implementada com *binding time* flexível. A *feature Notation* é do tipo OU (OR), pois ambas as configurações podem estar no mesmo produto. Portanto, os seguintes sistemas podem ser gerados: (i) somente notação UML 1.4 (estático); (ii) somente notação Java (estático); e (iii) notações Java e UML 1.4 (dinâmico).

Outro recurso que ArgoUML disponibiliza é o uso de *guillemets* (<< >>). Esse recurso também foi extraído e implementado com *binding time* flexível. A escolha da *feature Guillemets* ocorreu devido ao seu espalhamento entre as camadas do sistema, fazendo da mesma um exemplo interessante. A *feature Guillemets* é do tipo opcional. Por conseguinte, em um mesmo sistema, tem-se um exemplo de *feature* bem modularizada (*Notation*) e um exemplo de *feature* espalhada (*Guillemets*). Alguns exemplos da modularização das *features* e implementação do *binding time* flexível com os três idiomas podem ser encontrados no apêndice B.

4.5 - Análise

Nesta seção, serão apresentados os resultados da avaliação de cada idioma referente às questões propostas na Seção 4.2. Cada critério de avaliação deve responder a uma das perguntas introduzidas. São quatro critérios: clone, espalhamento, entrelaçamento e tamanho.

4.5.1 - Clone

A identificação de clones é importante, pois pode determinar a qualidade do código. Visando responder a Questão 1 da Seção 4.2, a métrica *Pairs of Clone Code* (PCC) foi utilizada. De maneira a representar melhor a clonagem de código, um critério foi definido. Somente foi considerado clone, dois pares de código com no mínimo quarenta *tokens* iguais. Ferramentas como o CCFinder [28] consideram clone dois pares de código com no mínimo cinqüenta *tokens* iguais. A razão pela qual o número de *tokens* similares foi diminuído se dá pelo fato de os resultados não terem sido representativos com cinqüenta *tokens*, pois PCC resultava zero. A tabela 4.5.1.1 exibe os resultados da métrica.

	AdviceExecution	**Edicts**	**Aspect Inheritance**
Icons	1	26	1
Clouds	0	27	3
Notation	1	1	1
Guillemets	5	30	10
NextPieceBox	0	0	0

Tabela 4.5.1.1 – Resultados da métrica PCC

Como foi corroborado anteriormente, Edicts tem o problema de duplicação de código da *feature* e do *driver*. Os índices de clonagem são altos especialmente para *features* com alta granularidade, que é o caso de *Icons*, *Clouds* e *Guillemets*. Nesses casos, o código da *feature* fica duplicado nos *advices* dos aspectos dinâmicos e estáticos. As *features Notation* e *NextPieceBox* são pequenas e bem modularizadas, por isso praticamente não há duplicação com Edicts. Aspect

Inheritance obteve baixos índices de duplicação, já que o código da *feature* não é clonado (uso do aspecto abstrato). A *feature Guillemets* obteve o maior índice devido ao espalhamento. Quando o número mínimo de *tokens* é diminuído, os índices de Aspect Inheritance aumentam consideravelmente por causa dos clones dos *drivers* como foi mostrado na Figura 3.2.1. Os melhores índices foram do idioma AdviceExecution, pois o código do *driver* está modularizado em um aspecto somente, e o código da *feature* não é duplicado.

4.5.2 – Espalhamento

Duas métricas foram responsáveis por responder a Questão 2 da Seção 4.2. CDC e DOS analisam o espalhamento de código. A análise foi dividida em duas partes importantes, a primeira é o espalhamento do código do *driver* e a segunda parte, do código da *feature*.

4.5.2.1 – *Driver*

AdviceExecution obteve os melhores índices de DOS. Como já foi citado, a implementação do *driver* fica encapsulada em um único aspecto. Edicts alcançou índices mais altos em relação ao AdviceExecution devido ao espalhamento das cláusulas condicionais nos *advices* dos aspectos que implementam código da *feature*, como os *if* mostrados na Figura 3.1.3. Aspect Inheritance obteve os piores índices (mais altos) em conseqüência das cláusulas condicionais definidas em todos os sub-aspectos e da junção entre os *pointcuts* do aspecto abstrato com essas cláusulas, exemplificados na Figura 3.2.2. A Figura 4.5.2.1.1 exibe os resultados da métrica DOS.

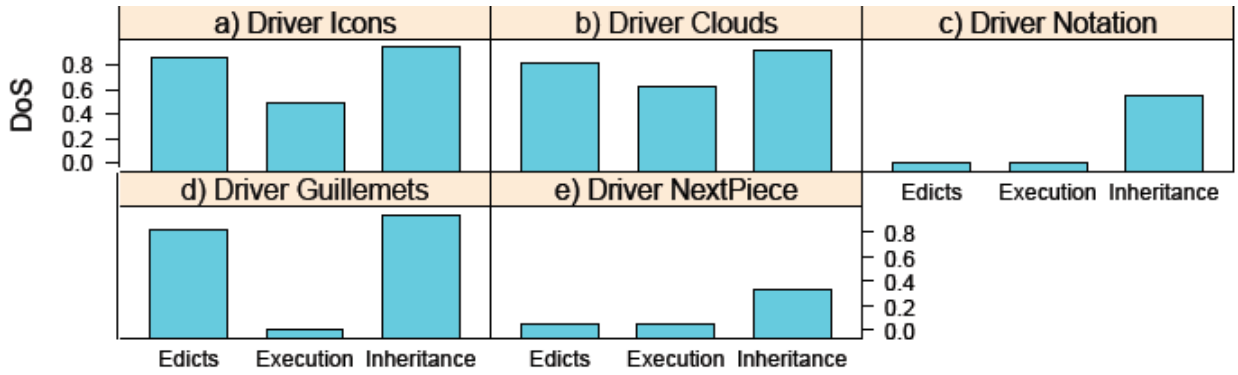

Figura 4.5.2.1.1 – Resultados da métrica DOS para o *driver*

Dois casos interessantes ocorreram com as *features Notation* e *NextPieceBox* em que os índices praticamente se equivalem entre Edicts e AdviceExecution. Ambas as *features* são pequenas e o código do *driver* só está localizado em um aspecto, diferentemente de Aspect Inheritance em que o *driver* se encontra espalhado em pelo menos dois aspectos. Outro caso curioso ocorre na *feature Guillemets*. O grau de espalhamento de AdviceExecution é muito inferior aos outros idiomas. Devido à alta granularidade de *Guillemets*, as cláusulas necessárias para introduzir o *driver* são bastante espalhadas entre os aspectos que modularizam a *feature*, enquanto que com AdviceExecution isso não ocorre.

Alguém pode questionar sobre os índices de AdviceExecution não serem iguais para todas as *features*, já que, como mencionado, toda a implementação do *driver* está localizada em somente um aspecto. Primeiro, é necessário perceber que as taxas são similares para *features* de um mesmo sistema, pois as implementações do *driver* são similares para um mesmo sistema. O que ocorre com *Clouds* e *Icons* é que a implementação do *driver* inicializa uma classe para leitura de um arquivo de propriedades, fato que acaba acarretando em um maior espalhamento do código do *driver*. Portanto, a implementação e o tipo do *driver* afeta diretamente a métrica DOS.

4.5.2.2 – *Feature*

Todos os idiomas obtiveram índices similares nas avaliações das *features*, com exceção de Edicts em *NextPieceBox*. Nesse caso, o código da *feature* está praticamente espalhado de forma igual entre os aspectos dinâmico e estático. Por outro lado, para AdviceExecution e Aspect Inheritance, o DOS é zero, visto que todo o código da *feature* está localizado em um único super-aspecto. A Figura 4.5.2.2.1 mostra os resultados.

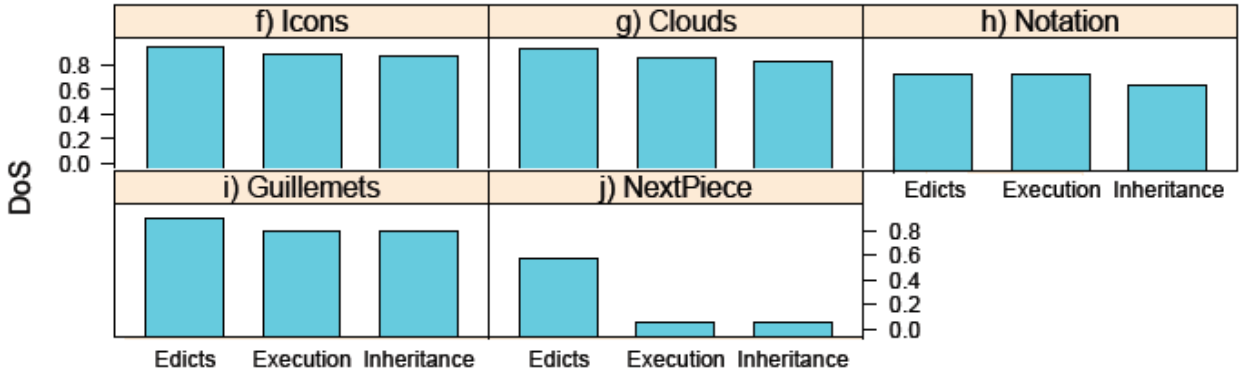

Figura 4.5.2.2.1 – Resultados da métrica DOS para a *feature*

DOS não foi suficiente para responder a questão sobre qual idioma pode reduzir o espalhamento, a figura 4.5.2.2.1 demonstra que todos os resultados foram similares, com exceção da *feature NextPieceBox*, por isso outra métrica foi utilizada. CDC permite verificar o espalhamento sob outra perspectiva. Levando em conta como exemplo a *feature Icons*, os resultados da métrica CDC são expressos na Tabela 4.5.2.2.2.

Edicts	AdviceExecution	Aspect Inheritance
19	10	9

Tabela 4.5.2.2.2 – Resultados da métrica CDC para *Icons*

Com base nos resultados apresentados na Tabela 4.5.2.2.2, pode-se afirmar que o idioma Edicts torna o trabalho de manutenção mais custoso. Portanto, as melhores técnicas para diminuir o espalhamento são: AdviceExecution e Aspect Inheritance.

4.5.3 – Entrelaçamento

Esta seção visa responder a Questão 3. Portanto, o entrelaçamento entre a *feature* e um *driver* é investigado. De acordo com o princípio de *separation of concerns*, um deve ser capaz de implementar e "cuidar" de ambos os *concerns* independentemente. Neste trabalho, é assumido que quanto maior for o entrelaçamento entre a *feature* e seu *driver*, menor é a separação dos *concerns*.

A métrica DOT [19] é usada para quantificar o entrelaçamento. Para que a análise seja melhor compreendida, uma nova métrica deve ser introduzida. *Lack of Degree of Tangling* (LDOT) é diretamente proporcional ao entrelaçamento (LDOT = 1 - DOT), quanto maior o LDOT, maior o entrelaçamento.

A Figura 4.5.3.1 mostra os índices da métrica LDOT com relação às interações entre *features* e *drivers*. Para a maioria das *features*, AdviceExecution obteve o menor entrelaçamento. Isso ocorre, pois todas as implementações dos *drivers* estão localizadas em aspectos únicos. Já Edicts obteve o maior entrelaçamento devido a sua implementação dos *drivers* que, de forma geral, estão espalhados por todos os aspectos dinâmicos que contêm código da *feature*, como mostrado na Figura 3.1.3.

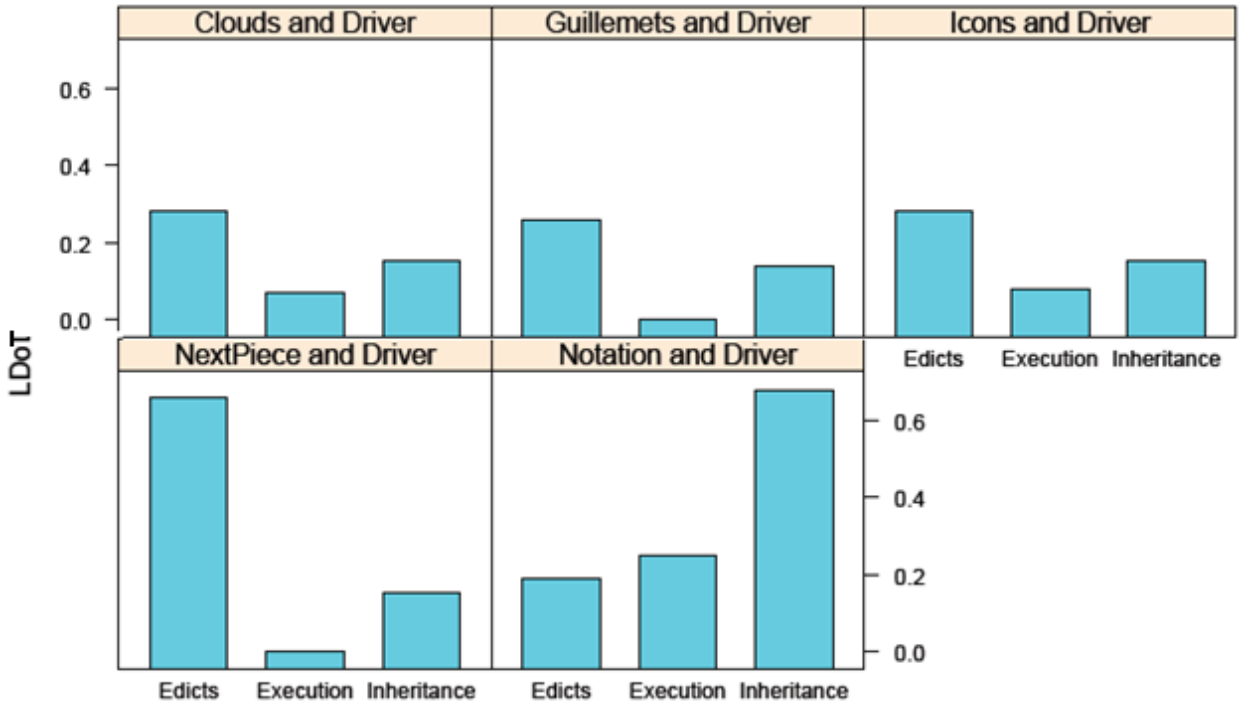

Figura 4.5.3.1 – Resultados da avaliação com a métrica LDOT

Casos interessantes foram observados em duas *features* avaliadas: *NextPieceBox* e *Notation*. A implementação de Edicts da primeira revela uma forte ausência de foco. Esse resultado ocorreu porque um único aspecto (*DynamicNextPieceBox*) implementa o comportamento de *NextPieceBox* e do *driver*, que na verdade é a interface com o usuário, que permite o mesmo habilitar a *feature NextPieceBox*. Todas as linhas de código desta interface

foram designadas ao *concern* do *driver*, que conduz a uma forte ausência de foco para o aspecto *DynamicNextPieceBox*.

Para a *feature Notation*, a implementação de Aspect Inheritance obteve o maior entrelaçamento registrado, em decorrência do *driver* que está espalhado nas implementações das opções de *Notation*. Essas opções (UML 1.4 e Java) são implementadas como sub-aspectos e têm seus *drivers* específicos. É importante observar que os sub-aspectos definem os *pointcuts* dos *drivers* concretos enquanto que o super-aspecto define o abstrato. Portanto, o forte entrelaçamento ocorre porque toda a hierarquia dos aspectos lida com o *driver*.

4.5.4 - Tamanho

A Questão 4 está relacionada com o tamanho de cada idioma em termos de linhas de código e número de componentes. As métricas SLOC e VS foram usadas. Os resultados da métrica SLOC são exibidos na Figura 4.5.4.1. Edicts é maior devido, novamente, a duplicação de código que ocorre nos aspectos dinâmicos e estáticos. Aspect Inheritance e AdviceExecution obtiveram índices similares. Os sub-aspectos de Aspect Inheritance possuem SLOC parecidos com os aspectos que implementam o *adviceexecution*.

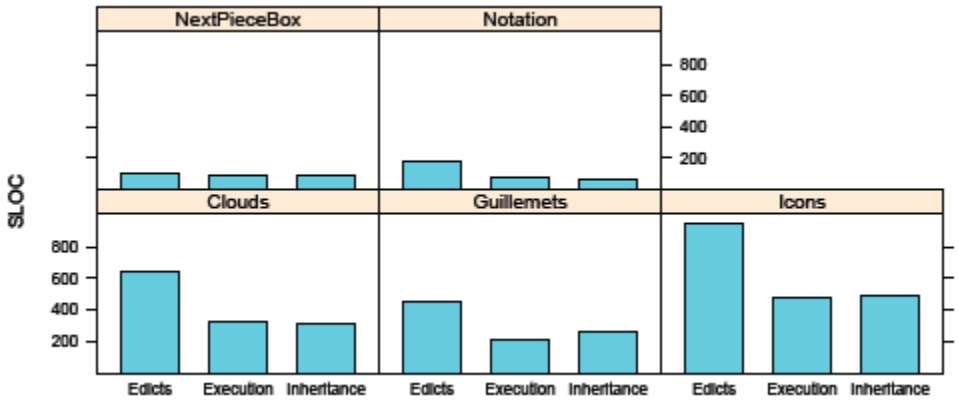

Figura 4.5.4.1 – Resultados da métrica SLOC

A análise baseada no número de componentes usa a métrica VS. Os resultados são exibidos na Figura 4.5.4.2. No geral, quando poucos aspectos são necessários para implementar a *feature*, todos os idiomas apresentam número similar de componentes. No entanto, a diferença entre AdviceExecution e os outros idiomas se torna mais visível, quando o número de aspectos usados na implementação é maior. AdviceExecution precisa de menos componentes, pois basta um aspecto para implementar cada *driver*. Edicts apresenta altos índices por causa da duplicação mencionada anteriormente. Aspect Inheritance apresentou os mais altos índices, o que indica que é o pior entre os três, pois precisa ter pelo menos dois sub-aspectos (dinâmico e estático) para cada aspecto que implementa código da *feature*.

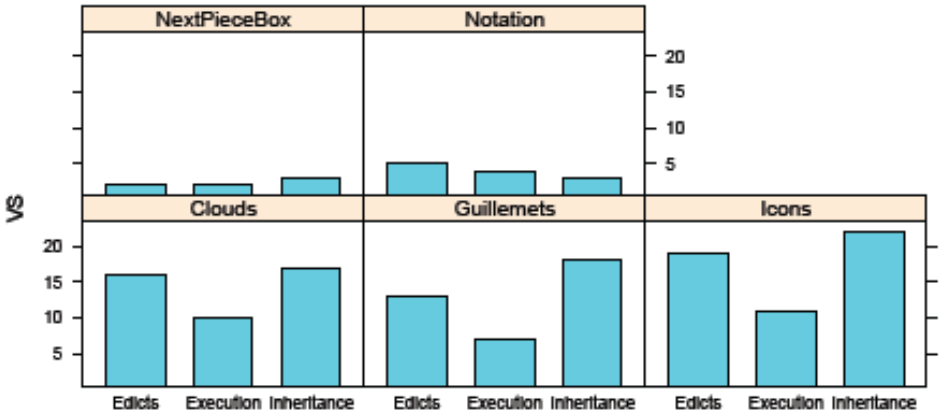

Figura 4.5.4.2 – Resultados da métrica VS

4.6 – Modelo para guiar desenvolvedores na escolha de idiomas

Nesta seção é apresentado um modelo para que desenvolvedores tenham sua tarefa de escolha do melhor idioma facilitada. Vários casos diferentes são abordados. O modelo funciona da seguinte maneira: dada uma *feature* implementada com AspectJ, deseja-se prover flexibilidade de *binding time* para ela. Ou seja, em alguns produtos a *feature* deve ser estática enquanto que em outros ela deve ser dinâmica. A partir disso, o desenvolvedor analisa os aspectos que implementam a *feature* e tenta combinar essa análise com algum dos casos do modelo. Se houver a combinação, o modelo sugere os idiomas. O modelo é introduzido na Figura 4.6.1.

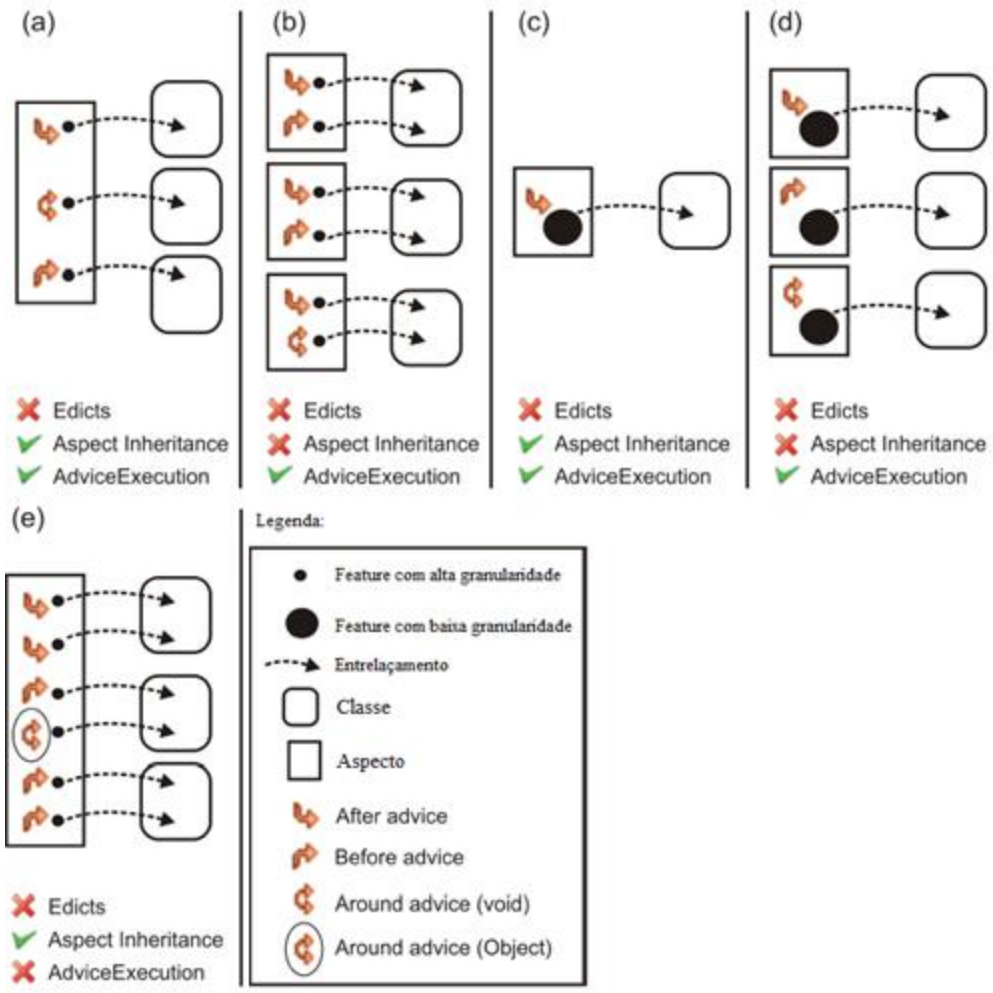

Figura 4.6.1 – Modelo para guiar desenvolvedores

A técnica Edicts não é recomendada em nenhum dos casos, pois o problema da duplicação de código ocorre em todas as situações.

O exemplo (a) consiste de um único aspecto que modulariza uma *feature* de alta granularidade. Aspect Inheritance pode ser aplicado, pois somente um aspecto é responsável por implementar a *feature*. No entanto, o desenvolvedor deve ter cuidado para os casos em que haja muitos *pointcuts*, pois isso aumenta o espalhamento do código do *driver*. Esses problemas não ocorrem com AdviceExecution, por isso também pode ser usado.

O exemplo (b) demonstra vários aspectos que modularizam uma *feature* de granularidade alta. Aspect Inheritance não é recomendado devido à necessidade de haver dois aspectos (dinâmico e estático) para cada aspecto que modulariza a *feature*. Portanto, AdviceExecution é recomendado para este caso.

O exemplo (c) define uma situação em que apenas um aspecto modulariza uma *feature* com baixa granularidade. Aspect Inheritance pode ser utilizado, pois apenas dois aspectos são necessários para implementar o código do *driver*. AdviceExecution também é recomendado; não

há duplicação de código e apenas um aspecto deve ser introduzido para implementar o código do *driver*. A recomendação do exemplo (c) é igual a do exemplo (a).

O exemplo (d) evidencia uma situação em que vários aspectos são utilizados para modularizar uma *feature* de baixa granularidade. Mais uma vez Aspect Inheritance não é aconselhado devido ao alto espalhamento do código do *driver*, já que há mais de um aspecto implementando a *feature*. AdviceExecution é o recomendado para essa situação. A recomendação do exemplo (d) é igual a do exemplo (b).

Por fim, o exemplo (e) simula uma situação em que apenas um aspecto é necessário e a *feature* possui alta granularidade. AdviceExecution não é recomendado, pois há um *around advice* que retorna um objeto. Como visto na Seção 3.3, não é possível usar AdviceExecution para essa situação. Aspect Inheritance é recomendado.

É importante citar que para o caso em que vários aspectos são definidos e que, por coincidência, possuam *around advices*, nenhum dos idiomas se aplica de maneira adequada. Desta forma, uma mistura de idiomas ou até mesmo o uso de outra técnica pode ser uma alternativa.

AdviceExecution é recomendado para a maioria dos casos, pois não há duplicação de código e os resultados das métricas são mais satisfatórios que os outros idiomas.

5 - Conclusões

Neste capítulo, apresentam-se as considerações finais sobre a análise feita em cima dos idiomas. Para concluir o trabalho, as limitações do estudo e trabalhos futuros são discutidos.

5.1 – Pontos estudados

Este trabalho teve como objetivo avaliar diferentes idiomas para implementação de *binding time* flexível de *features* em linhas de produto. A avaliação foi baseada no estudo de métricas conhecidas para análises quantitativas e qualitativas. Alguns aspectos importantes levados em conta foram a modularização de *features*, a duplicação, espalhamento e entrelaçamento de código.

O primeiro idioma estudado foi Edicts. Problemas como duplicação de código e espalhamento do *driver* foram encontrados e confirmados através dos resultados das métricas utilizadas neste trabalho. Tais problemas são prejudiciais para manutenção e tornam o sistema suscetível a erros, além do aumento de custos financeiro e de tempo de desenvolvimento [35].

O segundo idioma estudado foi Aspect Inheritance. Proposto para resolver alguns problemas de Edicts, esse idioma elimina a duplicação de código referente à *feature* com o conceito de herança em aspectos. Alguns pontos negativos foram observados como a criação de um aspecto exclusivo para *binding time* estático, que na verdade não faz sentido, já que não é necessário *driver* para *binding time* estático. Outro ponto negativo é que, devido a inexistência de herança múltipla em AspectJ, dois aspectos (para implementar o *driver*) têm que ser definidos para cada aspecto que encapsula o código da *feature*. Com isso, o código do *driver* fica espalhado, dificultando sua manutenção e a introdução de novos *drivers.*

O terceiro idioma estudado foi AdviceExecution. Esse idioma mostrou-se eficiente no contexto dos problemas apresentados pelos idiomas anteriores. A análise das métricas apresentou baixos índices de duplicação e espalhamento de código. No entanto, um ponto negativo foi observado. No caso específico em que há um *around advice* que retorna algum tipo de objeto, *null* pode ser retornado, o que pode provocar o lançamento de uma exceção em tempo de execução.

O foco do trabalho consistiu em analisar as vantagens e desvantagens de cada idioma em relação à modularidade provida. No entanto, para iniciar essa análise, primeiramente as *features* precisaram ser implementadas com *binding times* flexíveis em todos os idiomas. O sistema Tetris foi utilizado para exemplificar e auxiliar o entendimento dos idiomas. Os sistemas Freemind e ArgoUML foram analisados e suas *features* foram escolhidas de forma que o maior nível de representatividade fosse alcançado. Ou seja, procurou-se selecionar *features* de diferentes tipos, granularidades e tamanhos para tentar generalizar os resultados do estudo.

Um modelo para guiar desenvolvedores foi apresentado com o objetivo de facilitar a escolha do melhor idioma para determinada situação.

Entre as três técnicas analisadas, AdviceExecution obteve o melhor desempenho. Mas de toda forma não pode ser aplicada a todas as situações devido à limitação supracitada.

5.2 – Limitações e problemas

É importante observar que há limitações neste trabalho:

- Todas as *features* foram modularizadas com o uso de AspectJ, que é a linguagem mais popular para programação orientada a aspectos, porém existem outras linguagens como CaesarJ [27] que também podem ser usadas.
- Somente foram abordadas situações com um número de *drivers* limitados, mas nem sempre o cenário é esse. Muitos sistemas complexos necessitam de vários *drivers*. Por exemplo, em um avião comercial a *feature* "desligar piloto automático", deve ser habilitada baseada em vários sensores complexos que vão desde cálculos de velocidade do avião e do vento a altitude de vôo e temperatura externa.
- As *features* dos sistemas estudados foram escolhidas para obter um alto nível de representatividade do problema. Elas usam muitos recursos de AspectJ. Foram escolhidas *features* com alta e baixa granularidade e com implementações diferentes no contexto de modularização. Porém, não necessariamente todos os cenários foram considerados, e pode haver situações adversas, como *features* opcionais de baixa granularidade e *features* OR de alta granularidade, visto que poucas *features* foram modularizadas.
- Um número limitado de métricas foi usado, os problemas encontrados podem se mostrar mais graves com uma análise mais precisa.

5.3 – Trabalhos futuros

Um ponto importante a ser estudado é o uso de mecanismos diferentes para modularizar as *features* e implementar *binding time* flexível. O uso de CaesarJ pode ser considerado e estudos e avaliações podem ser feitas. Esta linguagem fornece mecanismos para fazer *deploy* de aspectos dinamicamente. Para fazer *deploy* do aspecto estaticamente é necessário que o mesmo possua uma palavra reservada em sua assinatura, ou então o aspecto deve ser instanciado e incluído no *build* para começar a funcionar.

A Figura 5.3.1 exibe um breve exemplo. Nesse caso, o *DynamicNextPieceBox* deve ter a palavra reservada enquanto que o aspecto responsável pelo código da *feature* (*NextPieceBox*) é incluído no *build* se a escolha do usuário for *Yes*.

Todo esse estudo pode ser feito usando CaesarJ, porém alguns problemas foram encontrados em testes extras. Recursos como compilação condicional, são necessários para inserir ou remover a palavra reservada.

```
1  public cclass NextPieceBox {
2     ...
3  }
4
5  public deployed cclass DynamicNextPieceBox {
6     ...
7     if (userChoice.equals("Yes")) {
8       deploy new NextPieceBox();
9     }
10    ...
11 }
```

Figura 5.3.1 – Exemplo de CaesarJ

Outro aspecto importante a ser analisado futuramente é a introdução de vários *drivers* de diferentes tipos, e verificar como os idiomas se comportam com relação à modularidade.

Todos os idiomas apresentados possuem limitações, portanto é o caso trabalhar em novas técnicas ou até em combinações das existentes. Mais estudos de caso podem ser considerados, ou até os mesmos estudos, porém com mais *features*, para melhorar a representatividade.

Referências

[1] J. Van Gurp, J. Bosch, M. Svahnberg: On the notion of variability in software product lines. Em *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Washington, DC, USA, IEEE Computer Society (2001).

[2] E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. Em *Proceedings of the 2003 Workshop on Software Variability Management (SVM)*, páginas 119–122, Gronigen, The Netherlands, Fevereiro 2003.

[3] M. Ribeiro, R. Cardoso, P. Borba, R. Bonifácio, H. Rebêlo. Does AspectJ Provide Modularity when Implementing Features with Flexible Binding Times?, (*LA-WASP'*09), Fortaleza-CE, Brasil, Outubro, 2009.

[4] E. Utrecht, G. Florijn, and E. Dolstra. Timeline variability: The variability of binding time of variation points. Em *Proceedings of the Workshop on Software Variability Management (SVM'03)*, páginas 119-122, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[6] E. Eide, A. Reid, J. Regehr, and J. Lepreau. Static and dynamic structure in design patterns. Em *Proceedings of the 24th International Conference. on Software Engineering (ICSE'02)*, páginas 208–218, Orlando, FL, Maio 2002.

[7] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: Implementing Features with Flexible Binding Times. Em *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, páginas 108-119, New York, NY, USA, 2008. ACM.

[8] J. Hannemann e G. Kiczales. Design pattern implementation in Java and AspectJ. Em *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, páginas 161–173, Seattle, WA, Nov. 2002.

[9] K. Pohl, G. Bockle, and F. J. van der Linden. Software Product Line Engineering. Springer, 2005.

[10] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison Wesley, 2001.

[11] V. Alves. Implementing Software Product Line Adoption Strategies. Tese PhD, Universidade Federal de Pernambuco, Recife, Brasil, Março 2007.

[12] M. Ribeiro. Restructuring Test Variabilities in Software Product Lines. Dissertação de Mestrado, Universidade Federal de Pernambuco, Recife, Brasil, Fevereiro 2008.

[13] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. Em *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, páginas 191-200, New York, NY, USA, 2006. ACM.

[14] P. Matos Jr. Analyzing techniques for implementing product line variabilities. Tese de Mestrado, Universidade Federal de Pernambuco, Recife, Brasil, 2008.

[15] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. *Technical Report (CMU/SEI-90 TR-21)*, Software Engineering Institute, Novembro 1990.

[16] P. Borba, S. Soares. Programação orientada a aspectos em Java. Universidade Federal de Pernambuco, Recife, Brasil.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, e W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, páginas 59–65.

[18] R. Bonifácio and P. Borba. Modeling scenario variability as crosscutting mechanisms. Em *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'09)*, páginas 125-136. ACM, 2009.

[19] M. Eaddy, A. Aho, and G. C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. Em *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07)*. IEEE Computer Society, 2007.

[20] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, e A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, páginas 497-515, 2008.

[21] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, e A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. Em *LNCS Transactions on Aspect-Oriented Software Development I*, páginas 36-74. Springer, 2006.

[22] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, e L. Bier. Clone detection using abstract syntax trees. Em *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, páginas 368-377, 1998.

[23] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, e M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, páginas 77-108, 1996.

[24] C. J. Kapser e M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, páginas 645-692, 2008.

[25] Freemind. Free mind mapping software, Outubro 2009. http://freemind.sourceforge.net/.

[26] ArgoUML. Argouml, Outubro 2009. http://argouml.tigris.org/.

[27] CaesarJ. CaesarJ Project, Outubro 2009. http://caesarj.org/.

[28] CCfinder Official Site, Outubro 2009. http://www.ccfinder.net/.

[29] P. Clements, L. M. Northrop . A Framework for Software Product Line Practice. Em *SEI Interactive*, Setembro 1999.

[30] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. A semantic web approach to feature modeling and verification. Em *Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Irlanda, Novembro de 2005.

[31] M. Anastasopoulos, e C. Gacek, (2001). Implementing Product Line Variabilities. Em *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*, páginas 109–117, New York, NY, USA. ACM Press.

[32] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier e J. Irwin. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1997.

[33] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can We Refactor Conditional Compilation into Aspects? Em *8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*, páginas 243–254.

[34] V. Basili, G. Caldeira, e H. D. Rombach, The Goal Question Metric Approach. Em *J. Marciniak (ed.), Encyclopedia of Software Engineering*, Wiley, 1994.

[35] M. Eaddy, T. Zimmerman, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, e A. V. Aho, Do Crosscutting Concerns Cause Defects? Em *IEEE Transactions on Software Engineering*, 2007.

Apêndices

Nesta seção, serão apresentados detalhes da modularização das *features* e da utilização dos idiomas. O código referente aos *drivers* está destacado com um contorno.

Apêndice A

Alguns exemplos extraídos do sistema Freemind são mostrados a seguir:

Edicts

```
1  public privileged aspect CloudActionAspect {
2
3      pointcut inicializarCloudAction(ControllerAdapter cthis)
4          : execution(* ControllerAdapter.inicializarCloudAction(..)) && this(cthis);
5
6      before(ControllerAdapter cthis) : inicializarCloudAction(cthis) {
7          String driver = new Driver().getProp().getProperty("driverCloud");
8          if (driver.equals("true")) {
9              cthis.cloud = new CloudAction(cthis);
10             cthis.cloudColor = new freemind.modes.actions.CloudColorAction(
11                     cthis);
12         }
13     }
14
15     pointcut enableCloud(ControllerAdapter cthis, boolean enabled)
16         : execution(* ControllerAdapter.enableCloud(..))
17             && this(cthis) && args(enabled);
18
19     before(ControllerAdapter cthis, boolean enabled) : enableCloud(cthis, enabled) {
20         String driver = new Driver().getProp().getProperty("driverCloud");
21         if (driver.equals("true")) {
22             cthis.cloudColor.setEnabled(enabled);
23         }
24     }
25 }
```

Figura 1 – Exemplo da *feature Clouds* com Edicts e *binding time* dinâmico

```
public privileged aspect CloudActionAspectStatic {
    pointcut inicializarCloudAction(ControllerAdapter cthis)
        : execution(* ControllerAdapter.inicializarCloudAction(..)) && this(cthis);

    before(ControllerAdapter cthis) : inicializarCloudAction(cthis) {
        cthis.cloud = new CloudAction(cthis);
        cthis.cloudColor = new freemind.modes.actions.CloudColorAction(cthis);
    }

    pointcut enableCloud(ControllerAdapter cthis, boolean enabled)
        : execution(* ControllerAdapter.enableCloud(..))
            && this(cthis) && args(enabled);

    before(ControllerAdapter cthis, boolean enabled) : enableCloud(cthis, enabled) {
        cthis.cloudColor.setEnabled(enabled);
    }
}
```

Figura 2 – Exemplo da *feature Clouds* com Edicts e *binding time* estático

```
public aspect NodeViewIcons {

    pointcut updateNodeView(NodeView nodeView):
        execution(void NodeView.update()) && this(nodeView);

    before(NodeView nodeView): updateNodeView(nodeView) {
        String driver = new Driver().getProp().getProperty("driverIcons");
        if (driver.equals("true")) {
            MultipleImage iconImages = new MultipleImage(nodeView.map.getZoom());
            boolean iconPresent = false;
            FreeMindMain frame = nodeView.map.getController().getFrame();
            Map stateIcons = (nodeView.getModel()).getStateIcons();
            for (Iterator i = stateIcons.keySet().iterator(); i.hasNext();) {
                String key = (String) i.next();
                iconPresent = true;
                ImageIcon myIcon = (ImageIcon) stateIcons.get(key);
                iconImages.addImage(myIcon);
            }

            List icons = (nodeView.getModel()).getIcons();
            for (Iterator i = icons.iterator(); i.hasNext();) {
                MindIcon myIcon = (MindIcon) i.next();
                iconPresent = true;
                iconImages.addImage(myIcon.getIcon(frame));
            }

            String link = ((NodeAdapter) nodeView.getModel()).getLink();
            if (link != null) {
                iconPresent = true;
                ImageIcon icon = new ImageIcon(frame.getResource(link
                        .startsWith("mailto:") ? "images/Mail.png" : (Tools
                        .executableByExtension(link) ? "images/Executable.png"
                        : "images/Link.png")));
                iconImages.addImage(icon);
            }

            nodeView.setIcon(iconPresent ? iconImages : null);
        }
    }
}
```

Figura 3 – Exemplo da *feature Icons* com Edicts e *binding time* dinâmico

```
public aspect NodeViewIconsStatic {
    pointcut updateNodeView(NodeView nodeView):
        execution(void NodeView.update()) && this(nodeView);

    before(NodeView nodeView): updateNodeView(nodeView) {
        MultipleImage iconImages = new MultipleImage(nodeView.map.getZoom());
        boolean iconPresent = false;
        FreeMindMain frame = nodeView.map.getController().getFrame();
        Map stateIcons = (nodeView.getModel()).getStateIcons();
        for (Iterator i = stateIcons.keySet().iterator(); i.hasNext();) {
            String key = (String) i.next();
            iconPresent = true;
            ImageIcon myIcon = (ImageIcon) stateIcons.get(key);
            iconImages.addImage(myIcon);
        }
        List icons = (nodeView.getModel()).getIcons();
        for (Iterator i = icons.iterator(); i.hasNext();) {
            MindIcon myIcon = (MindIcon) i.next();
            iconPresent = true;
            iconImages.addImage(myIcon.getIcon(frame));
        }
        String link = ((NodeAdapter) nodeView.getModel()).getLink();
        if (link != null) {
            iconPresent = true;
            ImageIcon icon = new ImageIcon(frame.getResource(link
                    .startsWith("mailto:") ? "images/Mail.png" : (Tools
                    .executableByExtension(link) ? "images/Executable.png"
                    : "images/Link.png")));
            iconImages.addImage(icon);
        }
        nodeView.setIcon(iconPresent ? iconImages : null);
    }
}
```

Figura 4 – Exemplo da *feature Icons* com Edicts e *binding time* estático

Aspect Inheritance

```
public privileged abstract aspect CloudActionAspect {

    abstract pointcut driver();

    pointcut inicializarCloudAction(ControllerAdapter cthis)
        : execution(* ControllerAdapter.inicializarCloudAction(..)) && this(cthis) && driver();

    pointcut enableCloud(ControllerAdapter cthis, boolean enabled)
        : execution(* ControllerAdapter.enableCloud(..))
            && this(cthis) && args(enabled) && driver();

    before(ControllerAdapter cthis) : inicializarCloudAction(cthis) {
        cthis.cloud = new CloudAction(cthis);
        cthis.cloudColor = new freemind.modes.actions.CloudColorAction(cthis);
    }

    before(ControllerAdapter cthis, boolean enabled) : enableCloud(cthis, enabled) {
        cthis.cloudColor.setEnabled(enabled);
    }
}
```

Figura 5 - Exemplo da *feature Clouds* com Aspect Inheritance, super-aspecto abstrato

```
1  public aspect CloudActionAspectDynamic extends CloudActionAspect {
2
3      pointcut driver() : if (new Driver().getProp()
4              .getProperty("driverCloud").equals("true"));
5  }
```

Figura 6 - Exemplo da *feature Clouds* com Aspect Inheritance, sub-aspecto dinâmico

```
1  public aspect CloudActionAspectStatic extends CloudActionAspect {
2
3      pointcut driver() : if (true);
4  }
```

Figura 7 - Exemplo da *feature Clouds* com Aspect Inheritance, sub-aspecto estático

```
1  public abstract aspect NodeViewIcons {
2
3      abstract pointcut driver();
4
5      pointcut updateNodeView(NodeView nodeView):
6          execution(void NodeView.update()) && this(nodeView) && driver();
7
8      before(NodeView nodeView): updateNodeView(nodeView) {
9          MultipleImage iconImages = new MultipleImage(nodeView.map.getZoom());
10         boolean iconPresent = false;
11         FreeMindMain frame = nodeView.map.getController().getFrame();
12         Map stateIcons = (nodeView.getModel()).getStateIcons();
13         for (Iterator i = stateIcons.keySet().iterator(); i.hasNext();) {
14             String key = (String) i.next();
15             iconPresent = true;
16             ImageIcon myIcon = (ImageIcon) stateIcons.get(key);
17             iconImages.addImage(myIcon);
18         }
19
20         List icons = (nodeView.getModel()).getIcons();
21         for (Iterator i = icons.iterator(); i.hasNext();) {
22             MindIcon myIcon = (MindIcon) i.next();
23             iconPresent = true;
24             iconImages.addImage(myIcon.getIcon(frame));
25         }
26
27         String link = ((NodeAdapter) nodeView.getModel()).getLink();
28         if (link != null) {
29             iconPresent = true;
30             ImageIcon icon = new ImageIcon(frame.getResource(link
31                     .startsWith("mailto:") ? "images/Mail.png" : (Tools
32                     .executableByExtension(link) ? "images/Executable.png"
33                     : "images/Link.png")));
34             iconImages.addImage(icon);
35         }
36
37         nodeView.setIcon(iconPresent ? iconImages : null);
38     }
39 }
```

Figura 8 - Exemplo da *feature Icons* com Aspect Inheritance, super-aspecto abstrato

```
1  public aspect NodeViewIconsDynamic extends NodeViewIcons {
2      pointcut driver() : if (new Driver().getProp()
3              .getProperty("driverIcons").equals("true"));
4  }
```

Figura 9 - Exemplo da *feature Icons* com Aspect Inheritance, sub-aspecto dinâmico

```
1  public aspect NodeViewIconsStatic extends NodeViewIcons {
2      pointcut driver() : if (true);
3  }
```

Figura 10 - Exemplo da *feature Icons* com Aspect Inheritance, sub-aspecto estático

AdviceExecution

```
1  public aspect CloudAdviceExecutionAspect {
2
3      Object around() : adviceexecution() && within(freemind.clouds.aspect.edicts.adviceexecution.*)
4          && !within(freemind.aspect.adviceexecution.CloudAdviceExecutionAspect) {
5          String driver = new Driver().getProp().getProperty("driverIcons");
6          if (driver.equals("true")) {
7              return proceed();
8          }
9          return null;
10     }
11 }
```

Figura 11 – Exemplo da *feature Clouds* com AdviceExecution, aspecto que implementa o *adviceexecution*

```
1  public privileged aspect CloudActionAspect {
2
3      pointcut inicializarCloudAction(ControllerAdapter cthis)
4          : execution(* ControllerAdapter.inicializarCloudAction(..)) && this(cthis);
5
6      before(ControllerAdapter cthis) : inicializarCloudAction(cthis) {
7          cthis.cloud = new CloudAction(cthis);
8          cthis.cloudColor = new freemind.modes.actions.CloudColorAction(cthis);
9      }
10
11     pointcut enableCloud(ControllerAdapter cthis, boolean enabled)
12         : execution(* ControllerAdapter.enableCloud(..))
13             && this(cthis) && args(enabled);
14
15     before(ControllerAdapter cthis, boolean enabled) : enableCloud(cthis, enabled) {
16         cthis.cloudColor.setEnabled(enabled);
17
18     }
19 }
```

Figura 12 – Exemplo da *feature Clouds* com AdviceExecution, aspecto que implementa código da *feature*

```
1  public aspect IconAdviceExecutionAspect {
2
3      Object around() : adviceexecution() && within(freemind.icons.aspect.edicts.adviceexecution.*)
4          && !within(freemind.icons.adviceexecution.IconAdviceExecutionAspect) {
5          String driver = new Driver().getProp().getProperty("driverIcons");
6          if (driver.equals("true")) {
7              return proceed();
8          }
9          return null;
10     }
11 }
```

Figura 13 - Exemplo da *feature Icons* com AdviceExecution, aspecto que implementa o *adviceexecution*

```
1  public aspect NodeViewIcons {
2
3      pointcut updateNodeView(NodeView nodeView):
4          execution(void NodeView.update()) && this(nodeView);
5
6      before(NodeView nodeView): updateNodeView(nodeView) {
7          MultipleImage iconImages = new MultipleImage(nodeView.map.getZoom());
8          boolean iconPresent = false;
9          FreeMindMain frame = nodeView.map.getController().getFrame();
10         Map stateIcons = (nodeView.getModel()).getStateIcons();
11         for (Iterator i = stateIcons.keySet().iterator(); i.hasNext();) {
12             String key = (String) i.next();
13             iconPresent = true;
14             ImageIcon myIcon = (ImageIcon) stateIcons.get(key);
15             iconImages.addImage(myIcon);
16         }
17
18         List icons = (nodeView.getModel()).getIcons();
19         for (Iterator i = icons.iterator(); i.hasNext();) {
20             MindIcon myIcon = (MindIcon) i.next();
21             iconPresent = true;
22             iconImages.addImage(myIcon.getIcon(frame));
23         }
24
25         String link = ((NodeAdapter) nodeView.getModel()).getLink();
26         if (link != null) {
27             iconPresent = true;
28             ImageIcon icon = new ImageIcon(frame.getResource(link
29                     .startsWith("mailto:") ? "images/Mail.png" : (Tools
30                     .executableByExtension(link) ? "images/Executable.png"
31                     : "images/Link.png")));
32             iconImages.addImage(icon);
33         }
34
35         nodeView.setIcon(iconPresent ? iconImages : null);
36     }
37 }
```

Figura 14 – Exemplo da *feature Icons* com AdviceExecution, aspecto que implementa código da *feature*

Apêndice B

Alguns exemplos extraídos do sistema ArgoUML são mostrados a seguir:

Edicts

```
1  public aspect ArgoUMLGuillemetsEventAspect {
2
3      pointcut addListenerGuillemotsHook(ExplorerEventAdaptor cthis)
4          : execution(* ExplorerEventAdaptor.addListenerGuillemotsHook(..))
5           && this(cthis);
6
7      before(ExplorerEventAdaptor cthis)
8              : addListenerGuillemotsHook(cthis) {
9          if (DriverGuillemets.getDriver().equals("true")) {
10             Configuration.addListener(Notation.KEY_USE_GUILLEMOTS, cthis);
11         }
12     }
13
14     pointcut isChangePropertyGuillemetsHook(final PropertyChangeEvent pce)
15         : execution(* ExplorerEventAdaptor.isChangePropertyGuillemetsHook(..))
16         && args(pce);
17
18     boolean around(final PropertyChangeEvent pce)
19         : isChangePropertyGuillemetsHook(pce) {
20         if (DriverGuillemets.getDriver().equals("true")) {
21             return Notation.KEY_USE_GUILLEMOTS.isChangedProperty(pce);
22         }
23         return false;
24     }
25 }
```

Figura 15 – Exemplo da *feature Guillemets* com Edicts e *binding time* dinâmico

```
1  public aspect ArgoUMLGuillemetsEventAspectStatic {
2
3      pointcut addListenerGuillemotsHook(ExplorerEventAdaptor cthis)
4          : execution(* ExplorerEventAdaptor.addListenerGuillemotsHook(..))
5           && this(cthis);
6
7      before(ExplorerEventAdaptor cthis)
8              : addListenerGuillemotsHook(cthis) {
9          Configuration.addListener(Notation.KEY_USE_GUILLEMOTS, cthis);
10     }
11
12     pointcut isChangePropertyGuillemetsHook(final PropertyChangeEvent pce)
13     : execution(* ExplorerEventAdaptor.isChangePropertyGuillemetsHook(..))
14      && args(pce);
15
16     boolean around(final PropertyChangeEvent pce)
17         : isChangePropertyGuillemetsHook(pce) {
18         return Notation.KEY_USE_GUILLEMOTS.isChangedProperty(pce);
19     }
20 }
```

Figura 16 – Exemplo da *feature Guillemets* com Edicts e *binding time* estático

```
1  public privileged aspect ArgoUMLNotationJavaUMLAspect {
2
3      public static String Notation.DEFAULT_NOTATION_NAME;
4      public static String Notation.DEFAULT_NOTATION_VERSION;
5      public static Icon Notation.ICON_NOTATION;
6      String driver = config.getValue("argo.notation.default", "");
7
8      public static IConfigurationFactory getFactory() {
9          return ConfigurationFactory.getInstance();
10     }
11     private static ConfigurationHandler config = getFactory()
12             .getConfigurationHandler();
13
14     before() : staticinitialization(Notation) {
15         if (driver.equals("UML 1.4")) {
16             Notation.DEFAULT_NOTATION_NAME = "UML";
17             Notation.DEFAULT_NOTATION_VERSION = "1.4";
18             Notation.ICON_NOTATION = ResourceLoaderWrapper
19                     .lookupIconResource("UmlNotation");
20         }
21         if (driver.equals("Java")) {
22             Notation.DEFAULT_NOTATION_NAME = "Java";
23             Notation.DEFAULT_NOTATION_VERSION = "";
24             Notation.ICON_NOTATION = ResourceLoaderWrapper
25                     .lookupIconResource("JavaNotation");
26         }
27     }
28     pointcut initNotationUmlHook() : execution(* Main.initNotationUmlHook());
29     before() : initNotationUmlHook() {
30         if (driver.equals("UML 1.4")) {
31             SubsystemUtility.initSubsystem(new InitNotationUml());
32         }
33     }
34     pointcut initNotationJavaHook() : execution(* Main.initNotationJavaHook());
35     before() : initNotationJavaHook() {
36         if (driver.equals("Java")) {
37             SubsystemUtility.initSubsystem(new InitNotationJava());
38         }
39     }
40 }
```

Figura 17 – Exemplo da *feature Notation* com Edicts e *binding time* dinâmico

```
1  public privileged aspect ArgoUMLNotationJavaUMLAspect {
2
3      public static String Notation.DEFAULT_NOTATION_NAME = "UML";
4
5      public static String Notation.DEFAULT_NOTATION_VERSION = "1.4";
6
7      public static Icon Notation.ICON_NOTATION = ResourceLoaderWrapper
8          .lookupIconResource("UmlNotation");
9
10     String driver = config.getValue("argo.notation.default", "");
11
12     public static IConfigurationFactory getFactory() {
13         return ConfigurationFactory.getInstance();
14     }
15
16     private static ConfigurationHandler config = getFactory()
17             .getConfigurationHandler();
18
19     pointcut initNotationUmlHook() : execution(* Main.initNotationUmlHook());
20
21     before() : initNotationUmlHook() {
22         SubsystemUtility.initSubsystem(new InitNotationUml());
23     }
24
25     pointcut initNotationJavaHook() : execution(* Main.initNotationJavaHook());
26
27     before() : initNotationJavaHook() {
28         SubsystemUtility.initSubsystem(new InitNotationJava());
29     }
30 }
```

Figura 18 – Exemplo da *feature Notation* com Edicts e *binding time* estático

Aspect Inheritance

```
1  public privileged abstract aspect ArgoUMLGuillemetsEventAspect {
2
3      abstract pointcut driver();
4
5      pointcut addListenerGuillemotsHook(ExplorerEventAdaptor cthis)
6          : execution(* ExplorerEventAdaptor.addListenerGuillemotsHook(..))
7           && this(cthis) && driver() ;
8
9      before(ExplorerEventAdaptor cthis)
10             : addListenerGuillemotsHook(cthis) {
11         Configuration.addListener(Notation.KEY_USE_GUILLEMOTS, cthis);
12     }
13
14     pointcut isChangePropertyGuillemetsHook(final PropertyChangeEvent pce)
15         : execution(* ExplorerEventAdaptor.isChangePropertyGuillemetsHook(..))
16         && args(pce) && driver();
17
18     boolean around(final PropertyChangeEvent pce)
19         : isChangePropertyGuillemetsHook(pce)  {
20         return Notation.KEY_USE_GUILLEMOTS.isChangedProperty(pce);
21     }
22
23 }
```

Figura 19 – Exemplo da *feature Guillemets* com Aspect Inheritance, super-aspecto abstrato

```
1  public privileged aspect ArgoUMLGuillemetsEventAspectDynamic extends
2          ArgoUMLGuillemetsEventAspect {
3      pointcut driver() : if (new DriverProperties()
4          .config.getValue("argo.notation.guillemots", "").equals("true"));
5  }
```

Figura 20 – Exemplo da *feature Guillemets* com Aspect Inheritance, sub-aspecto dinâmico

```
1  public aspect ArgoUMLGuillemetsEventAspectStatic
2      extends ArgoUMLGuillemetsEventAspect {
3
4      pointcut driver() : if (true);
5  }
```

Figura 21 – Exemplo da *feature Guillemets* com Aspect Inheritance, sub-aspecto estático

```
1  public privileged abstract aspect ArgoUMLNotationAspectInheritance {
2
3      public static String Notation.DEFAULT_NOTATION_NAME;
4      public static String Notation.DEFAULT_NOTATION_VERSION;
5      public static Icon Notation.ICON_NOTATION;
6
7      abstract pointcut driver();
8
9      public static IConfigurationFactory getFactory() {
10         return ConfigurationFactory.getInstance();
11     }
12
13     private static ConfigurationHandler config = getFactory()
14             .getConfigurationHandler();
15 }
```

Figura 22 – Exemplo da *feature Notation* com Aspect Inheritance, super-aspecto abstrato

```
1  public privileged aspect ArgoUMLNotationJavaAspectInheritance extends
2          ArgoUMLNotationAspectInheritance {
3
4      before() : staticinitialization(Notation) && driver() {
5          Notation.DEFAULT_NOTATION_NAME = "Java";
6          Notation.DEFAULT_NOTATION_VERSION = "";
7          Notation.ICON_NOTATION = ResourceLoaderWrapper
8                  .lookupIconResource("JavaNotation");
9      }
10
11     pointcut driver() : if (config.getValue("argo.notation.default", "").equals("Java"));
12
13     pointcut initNotationJavaHook() : execution(* Main.initNotationJavaHook()) && driver();
14
15     before() : initNotationJavaHook() {
16         SubsystemUtility.initSubsystem(new InitNotationJava());
17     }
18 }
```

Figura 23 – Exemplo da *feature Notation* com Aspect Inheritance, sub-aspecto Java

```
1  public privileged aspect ArgoUMLNotationUMLAspectInheritance extends
2          ArgoUMLNotationAspectInheritance {
3
4      before() : staticinitialization(Notation) && driver() {
5          Notation.DEFAULT_NOTATION_NAME = "UML";
6          Notation.DEFAULT_NOTATION_VERSION = "1.4";
7          Notation.ICON_NOTATION = ResourceLoaderWrapper
8                  .lookupIconResource("UmlNotation");
9      }
10
11     pointcut driver() : if (config.getValue("argo.notation.default", "").equals("UML 1.4"));
12
13     pointcut initNotationUmlHook() : execution(* Main.initNotationUmlHook()) && driver();
14
15     before() : initNotationUmlHook() {
16         SubsystemUtility.initSubsystem(new InitNotationUml());
17     }
18 }
```

Figura 24 – Exemplo da *feature Notation* com Aspect Inheritance, sub-aspecto UML

AdviceExecution

```
1  public aspect ArgoUMLGuillemetsAdviceExecutionAspect {
2
3      Object around() : adviceexecution()
4          && within(org.argouml.aspect.guillemets.adviceexecution.*)
5          && !within(org.argouml.aspect.guillemets.adviceexecution
6                  .ArgoUMLGuillemetsAdviceExecutionAspect) {
7          if (DriverGuillemets.getDriver().equals("true")) {
8              return proceed();
9          }
10         return null;
11     }
12 }
```

Figura 25 - Exemplo da *feature Guillemets* com AdviceExecution, aspecto que implementa o *adviceexecution*

```
1  public aspect ArgoUMLGuillemetsEventAspect {
2
3      pointcut addListenerGuillemotsHook(ExplorerEventAdaptor cthis)
4          : execution(* ExplorerEventAdaptor.addListenerGuillemotsHook(..))
5           && this(cthis);
6
7      before(ExplorerEventAdaptor cthis)
8              : addListenerGuillemotsHook(cthis) {
9          Configuration.addListener(Notation.KEY_USE_GUILLEMOTS, cthis);
10     }
11
12     pointcut isChangePropertyGuillemetsHook(final PropertyChangeEvent pce)
13     : execution(* ExplorerEventAdaptor.isChangePropertyGuillemetsHook(..))
14      && args(pce);
15
16     boolean around(final PropertyChangeEvent pce)
17         : isChangePropertyGuillemetsHook(pce) {
18         return Notation.KEY_USE_GUILLEMOTS.isChangedProperty(pce);
19     }
20 }
```

Figura 26 – Exemplo da *feature Guillemets* com AdviceExecution, aspecto que implementa código da *feature*

```
public aspect ArgoUMLNotationAspectAdviceExecution {

    public static IConfigurationFactory getFactory() {
        return ConfigurationFactory.getInstance();
    }

    private static ConfigurationHandler config = getFactory()
            .getConfigurationHandler();

    Object around() : adviceexecution()
        && within(org.argouml.aspect.adviceexecution.ArgoUMLNotationUmlAspectAdviceExecution) {
        if (config.getValue("argo.notation.default", "").equals("UML 1.4")) {
            return proceed();
        }
        return null;
    }

    Object around() : adviceexecution()
        && within(org.argouml.aspect.adviceexecution.ArgoUMLNotationJavaAspectAdviceExecution) {
        if (config.getValue("argo.notation.default", "").equals("Java")) {
            return proceed();
        }
        return null;
    }
}
```

Figura 27 - Exemplo da *feature Notation* com AdviceExecution, aspecto que implementa o *adviceexecution*

```
public abstract aspect ArgoUMLNotationJavaUmlAbstractAspectAdviceExecution {

    public static String Notation.DEFAULT_NOTATION_NAME;

    public static String Notation.DEFAULT_NOTATION_VERSION;

    public static Icon Notation.ICON_NOTATION;
}
```

Figura 28 - Exemplo da *feature Notation* com AdviceExecution, aspecto código comum a Java e UML 1.4

```
public privileged aspect ArgoUMLNotationJavaAspectAdviceExecution extends
        ArgoUMLNotationJavaUmlAbstractAspectAdviceExecution {

    before() : staticinitialization(Notation) {
        Notation.DEFAULT_NOTATION_NAME = "Java";
        Notation.DEFAULT_NOTATION_VERSION = "";
        Notation.ICON_NOTATION = ResourceLoaderWrapper
                .lookupIconResource("JavaNotation");
    }

    pointcut initNotationJavaHook() : execution(* Main.initNotationJavaHook());

    before() : initNotationJavaHook() {
        SubsystemUtility.initSubsystem(new InitNotationJava());
    }
}
```

Figura 29 - Exemplo da *feature Notation* com AdviceExecution, aspecto referente a Java

```
1  public privileged aspect ArgoUMLNotationUmlAspectAdviceExecution extends
2          ArgoUMLNotationJavaUmlAbstractAspectAdviceExecution {
3
4      before() : staticinitialization(Notation) {
5          Notation.DEFAULT_NOTATION_NAME = "UML";
6          Notation.DEFAULT_NOTATION_VERSION = "1.4";
7          Notation.ICON_NOTATION = ResourceLoaderWrapper
8                  .lookupIconResource("UmlNotation");
9      }
10
11     pointcut initNotationUmlHook() : execution(* Main.initNotationUmlHook());
12
13     before() : initNotationUmlHook() {
14         SubsystemUtility.initSubsystem(new InitNotationUml());
15     }
16 }
```

Figura 30 - Exemplo da *feature Notation* com AdviceExecution, aspecto referente a UML